

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE ENERGIAS ALTERNATIVAS E RENOVÁVEIS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Jaedson Barbosa Serafim

Monitoramento de pressão e vazão utilizando Internet das Coisas e computação em nuvem

João Pessoa

2023

Jaedson Barbosa Serafim

Monitoramento de pressão e vazão utilizando Internet das Coisas e computação em nuvem

Trabalho de Conclusão de Curso apresentado à Universidade Federal da Paraíba como exigência para a obtenção do título de Bacharel em Engenharia Elétrica.

Universidade Federal da Paraíba
Centro de Energias Alternativas e Renováveis
Curso de Graduação em Engenharia Elétrica

Orientador: Prof. Juan M. Maurício Villanueva

João Pessoa

2023

Jaedson Barbosa Serafim

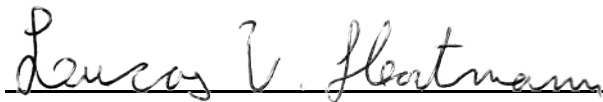
Monitoramento de pressão e vazão utilizando internet das coisas e computação em nuvem

Trabalho de Conclusão de Curso apresentado à Universidade Federal da Paraíba como exigência para a obtenção do título de Bacharel em Engenharia Elétrica.


Trabalho aprovado. João Pessoa, 26 de outubro de 2023:



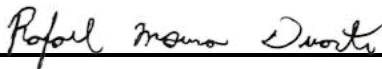
Prof. Juan M. Mauricio Villanueva
Orientador



Prof. Dr. Lucas Vinicius Hartmann
Examinador

 **ALTAMARALENCAR CARDOSO**
Data: 03/11/2023 12:00:23-0300
Verifique em <https://validar.it.gov.br>

Altamar Alencar Cardoso
Examinador



Rafael Moura Duarte
Convidado 3

João Pessoa
2023

Agradecimentos

Agradeço primeiramente a Deus, pela minha vida, e por me ajudar a ultrapassar todos os obstáculos encontrados ao longo do curso.

À todos os professores que me educaram e instruíram durante minha jornada.

Aos meus pais, pelo apoio indescritível e constante durante toda a minha vida.

Aos meus primos e tios, Josinês e Amilton, por terem me acolhido em sua casa durante toda a graduação.

À todos os demais familiares por todo o apoio, amparo e compreensão.

Aos meus colegas e amigos de curso por todos os momentos que compartilhamos. Em especial Aoliabe, Dominique, Gustavo, Jean, João, Laís, Marcelo e Zenivaldo.

Ao professor Juan Maurício Villanueva, por ter sido um orientador extremamente solícito.

Ao CNPQ pela bolsa para o desenvolvimento deste trabalho; à AWS pelos créditos; e à CAGEPA pelos hidrômetros fornecidos e pelas reuniões junto à Altamar, que auxiliaram no alinhamento da pesquisa.

Por fim, a todos os demais que contribuíram direta ou indiretamente para a realização deste trabalho.

Resumo

Em todo sistema hidráulico é importante que seja feito o controle da vazão e pressão do sistema. Em muitos casos ainda se usa aparelhos analógicos por causa dos custos envolvidos numa modernização. Entretanto, o monitoramento de dados de interesse é uma forma eficaz de combater o desperdício de água, recurso escasso que precisa ser consumido de forma responsável. Uma solução consiste em criar um sistema de telemetria com Internet das Coisas de baixo custo para viabilizar a instalação de sensores em vários pontos da rede de distribuição e conseguir acessar os dados pela Internet. Para demonstrar esta solução, este trabalho propõe um sistema de bateria e conversores capaz de alimentar e ler sensores de vazão e pressão a um baixo custo. Um microcontrolador ESP32 foi utilizado para controlar os sensores, a carga da bateria e registrar todas as variáveis importantes. Através do protocolo MQTT os dados foram lançados para uma plataforma online onde podem ser acessados e salvos conforme desejado. Os resultados experimentais comprovam a eficácia do sistema instalado para acessar a informação localmente ou via Internet, assim como sua confiabilidade.

Palavras-chave: Inteligência artificial, Internet das Coisas, Rust, ESP32.

Abstract

In every hydraulic system, it is important to control the flow and pressure of the system. In many cases, analog devices are still used because of the costs involved in modernization. However, monitoring data of interest is an effective way to combat water waste, a scarce resource that needs to be consumed responsibly. One solution consists of creating a low-cost telemetry system with the Internet of Things to enable the installation of sensors at various points in the distribution network and access data via the Internet. To demonstrate this solution, this work proposes a battery and converter system capable of powering and reading flow and pressure sensors at a low cost. An ESP32 microcontroller was used to control the sensors, battery charge and record all important variables. Through the MQTT protocol, data is released to an online platform where it can be accessed and saved as desired. The experimental results illustrate the effectiveness of the installed system to access information locally or via the internet, as well as its reliability.

Keywords: Artificial intelligence, Internet of Things, Rust, ESP32.

Lista de ilustrações

Figura 1 – Índice de perdas na distribuição de água por macrorregião geográfica . . .	17
Figura 2 – Logotipo da linguagem de programação Rust.	19
Figura 3 – Sensor de pressão usado.	25
Figura 4 – Sensor de vazão usado	26
Figura 5 – Laboratório de Eficiência Energética e Hidráulica em Saneamento (LE-NHS).	27
Figura 6 – Hardware desenvolvido	29
Figura 7 – Diagrama do circuito elevador de tensão	30
Figura 8 – Diagrama do circuito de energia	31
Figura 9 – Saída da geração de projeto	34
Figura 10 – Estrutura inicial de arquivos do projeto	34
Figura 11 – Diagrama geral do sistema.	37
Figura 12 – Estrutura de arquivos do aplicativo Web de monitoramento	40
Figura 13 – Captura de tela do aplicativo de monitoramento	41
Figura 14 – Diagrama simplificado de comunicação.	42
Figura 15 – Instalação do protótipo de monitoramento no LENHS	43
Figura 16 – Local de instalação do hardware do projeto e do repetidor	44
Figura 17 – Tensão na bateria (mV)	47
Figura 18 – Frequência de pulsos do sensor de vazão (Hz)	47
Figura 19 – Tensão na bateria (mV)	48
Figura 20 – Frequência de pulsos do sensor de vazão (Hz)	48
Figura 21 – Tensão sobre o sensor de pressão	49
Figura 22 – Tensão sobre o transistor NPN	50
Figura 23 – Tensão sobre o resistor do sensor de pressão	50
Figura 24 – Tensão sobre o transistor durante ligamento do sensor de pressão . . .	51
Figura 25 – Tensão sobre o transistor após estabilização do sensor de pressão . . .	51
Figura 26 – Tensões da bateria e na entrada do LDO durante 1º teste	52
Figura 27 – Tensão lida sobre resistor do sensor de pressão durante 1º teste	52
Figura 28 – Tensão da bateria e na entrada do LDO durante 2º teste	53
Figura 29 – Tensão lida sobre resistor do sensor de pressão durante 2º teste	53
Figura 30 – Estimativa de porcentagem de bateria com validação no 1º teste	54
Figura 31 – Estimativa de porcentagem de bateria com validação no 2º teste	54

Lista de abreviaturas e siglas

LENHS	Laboratório de Eficiência Energética e Hidráulica em Saneamento
SNIS	Sistema Nacional de Informações sobre Saneamento
IoT	Internet of Things
IA	Inteligência artificial
MQTT	Message Queuing Telemetry Transport
ADC	Conversor analógico-digital

Sumário

1	INTRODUÇÃO	17
1.1	Pertinência e motivação do trabalho	17
1.2	Objetivos	18
1.3	Organização do trabalho	18
2	FUNDAMENTAÇÃO TEÓRICA	19
2.1	Rust	19
2.2	Software embarcado e IoT	20
2.3	Protocolos MQTT e HTTP	20
2.4	JSON e CSV	21
2.5	Programação Web básica	22
2.6	Programação Web moderna	22
2.7	Conversor CC-CC	23
2.7.1	Conversor Boost	23
2.7.2	Conversor SEPIC	24
2.8	Rede de abastecimento de água	24
2.9	Sensores de vazão e pressão	24
3	MATERIAIS E MÉTODOS	27
3.1	Materiais Utilizados	28
3.2	Desenvolvimento do hardware	29
3.2.1	Elevador de tensão	30
3.2.2	Bateria e gestão de energia	31
3.2.3	Integração com ADC e ESP	32
3.3	Desenvolvimento do <i>software</i> embarcado	33
3.3.1	Pré-ambiente de desenvolvimento	33
3.3.2	Criação de projeto e estrutura de arquivos	33
3.3.3	Bibliotecas utilizadas	35
3.3.4	Variáveis de ambiente e constantes	36
3.3.5	Diagrama geral	36
3.4	Desenvolvimento do servidor	38
3.4.1	Criação e bibliotecas	38
3.4.2	Código desenvolvido	38
3.5	Desenvolvimento do aplicativo de monitoramento	39
3.6	Comunicação	42
3.7	Monitoramento contínuo de pressão e vazão	43

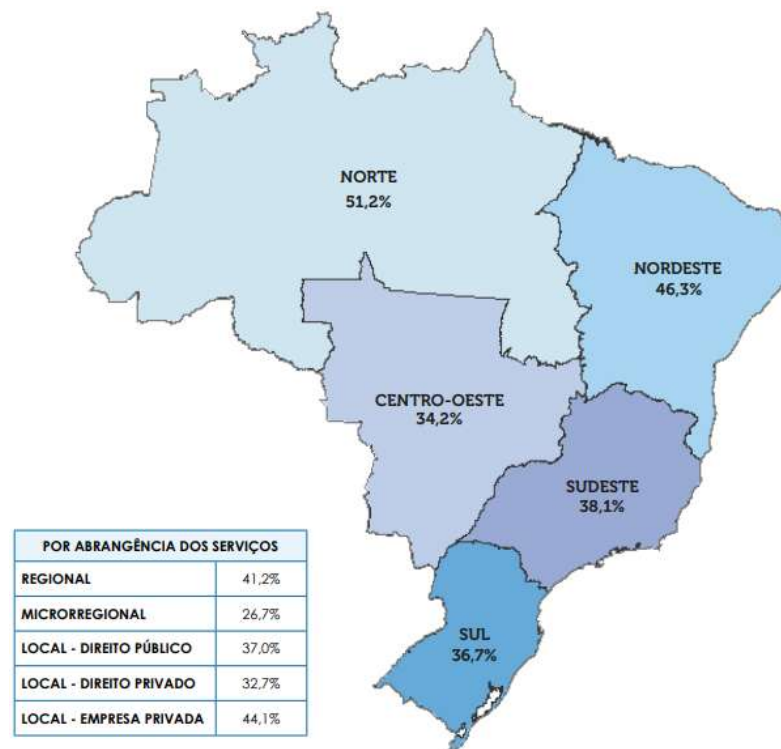
3.8	Análise do sensor de pressão com conversor <i>boost</i>	45
3.9	Análise de autonomia de bateria	45
3.10	Estimação de carga de bateria em uso com IA	46
4	RESULTADOS	47
4.1	Monitoramento contínuo de vazão e pressão	47
4.2	Análise do sensor de pressão com conversor <i>boost</i>	49
4.3	Análise de autonomia de bateria	52
4.4	Estimativa de porcentagem com IA	54
5	CONCLUSÃO	55
	REFERÊNCIAS	57
	 ANEXOS	 59
	ANEXO A – MANUAL DO TRANSDUTOR DE PRESSÃO	61
	ANEXO B – EXEMPLO DE PÁGINA WEB	65
	ANEXO C – EXEMPLO DE CÓDIGO SVELTE	67
	ANEXO D – CONFIG.TOML DO PROJETO EMBARCADO	69
	ANEXO E – CÓDIGO EMBARCADO NO ESP	71
	ANEXO F – CÓDIGO DO SERVIDOR	83
	ANEXO G – CÓDIGO DO APLICATIVO WEB	85
	ANEXO H – CÓDIGO DE TREINAMENTO DA IA	89

1 Introdução

1.1 Pertinência e motivação do trabalho

No Brasil, o índice de perdas na distribuição de água é de 40,1% (Brasil. SNIS, 2021, p.-38). Essa porcentagem é diferente para cada uma das macrorregiões geográficas, como ilustrado na Figura 1, e tais perdas são divididas em: perda aparente, quando a água não é contabilizada por causa de ligações clandestinas e submedição; e perda real, quando há vazamentos em pontos das infraestruturas de distribuição.

Figura 1 – Índice de perdas na distribuição de água por macrorregião geográfica



Fonte: SNIS

Atualmente as concessionárias responsáveis pelo abastecimento de água utilizam hidrômetros analógicos para a medição do consumo de cada usuário, sendo necessário o deslocamento de um leiturista até cada unidade consumidora para a coleta das medições e, a partir disso, emitir a fatura do cliente. Isso impossibilita, por exemplo, um alerta em tempo real caso ocorra uma medição inconsistente, como em caso de vazamento dentro das instalações do consumidor (OLIVEIRA *et al.*, 2022).

Para a redução das perdas reais é importante um controle de pressão eficaz, por influenciar o número e a vazão dos vazamentos, além de contribuir para a redução do

consumo energético do sistema, uma vez que o conjunto motor-bomba não necessita operar a plena carga durante todo o dia. Nesse contexto, um grande desafio para esse controle é a baixa quantidade de sensores e sua distância para os atuadores do processo, sendo necessário o emprego de tecnologias IoT (*Internet of Things* ou Internet da Coisas) para a comunicação desses a um baixo custo (ARAÚJO, 2021).

Caso sejam aplicados medidores eletrônicos nas unidades consumidores e ampliado o sensoriamento em pontos estratégicos do sistema de distribuição, será possível comparar o volume enviado para uma região e o consumo de água. Deste modo, em caso de diferenças, pode-se combater os focos de perdas em zonas mais restritas das cidades, identificar falta de água ou baixa pressão em postos da rede, bem como elaborar políticas de consumo consciente da água, como tarifa horária (VENDEMIATTI, 2020).

1.2 Objetivos

Este trabalho tem como objetivo desenvolver uma solução de monitoramento de redes hidráulicas de baixo custo compatível tanto com sensores dos macromedidores das rede de distribuição, quanto aos hidrômetros pulsados residenciais instalados hoje em milhões de residências pelo Brasil. Foi construído um protótipo para a coleta e transmissão de dados de pressão e vazão pela Internet a um servidor central, de onde tais informações podem ser acessadas a partir de uma plataforma Web. Assim, pretende-se reduzir o tempo para identificação de vazamentos e minimizar seus impactos financeiros e ambientais.

Os quatro objetivos específicos deste trabalho são: desenvolver conversor *boost* eficiente para alimentar sensor de pressão; monitorar pressão e vazão usando *IoT* e computação em nuvem; avaliar eficiência energética da coleta e transmissão de dados; e, por fim, analisar o *SOC* (estado de carga) da bateria usando IA (Inteligência Artificial).

1.3 Organização do trabalho

Além deste capítulo introdutório, este trabalho é composto por mais quatro: o segundo apresenta a fundamentação teórica necessária para compreender a realização projeto. Conceitos relacionados a sistemas de abastecimento de água, sensores de pressão e vazão, Internet das coisas e programação em Rust e de sistemas embarcados são discutidos e exemplificados; o terceiro aborda os materiais e métodos utilizados, partindo da descrição do sistema com uma visão geral descrevendo o que foi implementado, tanto no tocante ao *hardware* para leitura dos sensores, como do *software* de todos os sistemas desenvolvidos; no Capítulo 4, apresentam-se os resultados coletados; e, por fim, no Capítulo 5 constam as conclusões deste trabalho, bem como propostas de trabalhos futuras relacionadas a todas as melhorias que podem ser ainda implementadas ao que já foi desenvolvido.

2 Fundamentação teórica

Nesta seção, explicam-se os conceitos e tecnologias aplicados neste projeto a fim de facilitar o entendimento dos próximos capítulos.

2.1 Rust

Rust é uma linguagem de programação multiparadigma compilada desenvolvida pela Mozilla Research, projetada para ser segura, concorrente e prática sem usar o *garbage collector* (coletor de lixo) como é feito pela maioria das linguagens, o que faz com que ela seja considerada a linguagem de programação mais admirada de acordo com todas as pesquisas do Stack Overflow feitas nos últimos 8 anos (OVERFLOW, 2023).

Rust significa enferrujado em português, o que explica o seu logotipo ser um R enferrujado, como mostrado na Figura 2, mas seu nome tem outra origem, ele é originário do fungo Pucciniales, que é um fungo, segundo Graydon Hoare, robusto, distribuído e "paralelo", características presentes desta nova linguagem que ele estava criando (KIBWEN, 2014).

Em Rust não existem ponteiros nulos ou ponteiros soltos, impossibilitando falhas de segmentação. A linguagem impede condição de corridas entre threads pois não é possível que duas *threads* possam modificar um mesmo valor ao mesmo tempo e apenas referências de somente leitura podem ser compartilhadas entre várias *threads*.

O sistema de enumerações dessa linguagem também é um dos mais avançados que existe e os dois *enums* mais usados são o *Option* e o *Result*, que representam respectivamente uma possível inexistência de valor e um possível erro e o correto tratamento das variáveis desse tipo permite criar um código "inquebrável", o que é simples de ser feito, já que a linguagem fornece diferentes formas de tratar cada um deles.

Figura 2 – Logotipo da linguagem de programação Rust.



Fonte: <https://www.rust-lang.org/>

Dois conceitos mais avançados da linguagem são o de programação genérica e o de *ownership* (posse), onde os valores só podem ter um dono; quando um dono sai do escopo, o valor é destruído; valores também podem ser emprestados (*borrowing*) ou movidos (*move*); e em certas ocasiões é necessário especificar o tempo de vida (*lifetime*) de uma referência. Estes conceitos são complexos mas os exemplos mostrados mais adiante no trabalho permitirão uma melhor compreensão deles.

2.2 Software embarcado e IoT

O *software* embarcado refere-se a programas de computador projetados para executar tarefas específicas em dispositivos eletrônicos dedicados. Esses dispositivos podem incluir microcontroladores, microprocessadores, DSPs (Processadores de Sinal Digital) e outros sistemas incorporados em máquinas e produtos do dia a dia. É otimizado para uso em *hardware* específico e é altamente eficiente em termos de recursos, pois muitas vezes opera com recursos limitados, como memória e processamento, por isso linguagens de programação comuns para desenvolver *software* embarcado incluem C e C++, devido à sua proximidade com o *hardware* e sua eficiência e, mais recentemente, Rust está sendo inserido nesse meio devido à sua grande segurança.

A IoT é uma rede de dispositivos físicos, veículos, eletrodomésticos e outros objetos incorporados com sensores, *software* e tecnologia de rede, permitindo a coleta e a troca de dados. Esses dispositivos podem se comunicar uns com os outros e com sistemas de gerenciamento centralizados. Ela aproveita várias tecnologias de conectividade, como Wi-Fi, Bluetooth, 3G, 4G, 5G, LoRa, e outras, para permitir que dispositivos se comuniquem e compartilhem informações.

A combinação de programação de *software* embarcado e IoT possibilita a criação de sistemas inteligentes que podem coletar dados em tempo real, tomar decisões autônomas e melhorar a eficiência em diversas áreas. Essas tecnologias desempenham um papel cada vez mais importante na transformação digital e na automação de processos em empresas e na vida cotidiana.

2.3 Protocolos MQTT e HTTP

O HTTP (*Hypertext Transfer Protocol*) e o MQTT (*Message Queuing Telemetry Transport*) são dois protocolos de comunicação amplamente utilizados em aplicações de Internet das Coisas (IoT).

O HTTP é o protocolo fundamental da World Wide Web, projetado para transferir recursos, como páginas da web, entre servidores e clientes (navegadores). Funciona no modelo de solicitação e resposta, onde um cliente (por exemplo, um navegador da web) faz

solicitações a um servidor para obter recursos (como páginas da web). Cada solicitação resulta em uma resposta do servidor. É adequado para aplicações web tradicionais, mas não é otimizado para dispositivos com recursos limitados.

O MQTT é um protocolo de mensagens leve projetado para a comunicação eficiente entre dispositivos em redes com largura de banda restrita e recursos limitados, comumente usados na IoT. Opera no modelo de publicação/assinatura (*publish/subscribe*), onde os dispositivos enviam mensagens para tópicos e outros dispositivos interessados em receber essas mensagens se inscrevem nesses tópicos. O MQTT é extremamente eficiente em termos de largura de banda e recursos de *hardware*. Ele é ideal para dispositivos com restrições de energia e conectividade, pois minimiza a sobrecarga de comunicação. Além disso, suporta QoS (*Quality of Service*) configurável para garantir a entrega confiável das mensagens.

Em resumo, o MQTT é uma escolha popular em aplicações de IoT devido à sua eficiência, baixa sobrecarga, suporte a conectividade intermitente e capacidade de lidar com comunicações assíncronas entre dispositivos. No entanto, é importante observar que o HTTP ainda é amplamente utilizado em cenários de IoT, especialmente quando a interoperabilidade com a web é uma consideração crítica. Portanto, a escolha entre MQTT e HTTP depende das necessidades específicas da aplicação de IoT.

2.4 JSON e CSV

JSON (*JavaScript Object Notation*) e CSV (*Comma-Separated Values*) são dois formatos de arquivo comuns usados para armazenar e compartilhar dados. Ambos têm suas características e são apropriados para diferentes finalidades.

O JSON é um formato de texto versátil que armazena dados em pares chave-valor, facilmente legível por humanos devido à sua estrutura organizada e identificação. Suporta tipos de dados como strings, números, booleanos, objetos, arrays e valores nulos e pode representar dados complexos de forma aninhada. É frequentemente usado em serviços web, armazenamento de configurações, banco de dados NoSQL e troca de dados entre aplicativos.

O CSV é um formato de texto simples que armazena dados em uma grade de valores separados por vírgulas (ou outro delimitador, como ponto e vírgula ou tabulação). Cada linha do arquivo representa um registro e as vírgulas separam os valores. É fácil de ler e escrever, tornando-o um formato comum para armazenar dados tabulares. No entanto, não é tão eficaz em armazenar dados complexos com estruturas aninhadas. Ele não tem suporte nativo para tipos de dados, portanto os valores são tratados como *strings* e a interpretação dos dados deve ser feita pelo aplicativo que lê o arquivo. É amplamente utilizado em planilhas, exportação/importação de dados, relatórios e análises de dados.

2.5 Programação Web básica

A programação de aplicações web com HTML, CSS e JavaScript é a base da construção de sites interativos e dinâmicos na web. Cada uma dessas tecnologias desempenha um papel fundamental na criação de experiências web ricas e funcionais.

HTML é a linguagem de marcação usada para criar a estrutura e o conteúdo de uma página web. Ele define elementos como cabeçalhos, parágrafos, links, imagens, formulários e muito mais. É composto por tags, que são elementos dentro de colchetes e elas são usadas para marcar o início e o fim de elementos HTML.

CSS é usado para estilizar e formatar elementos HTML. Ele permite controlar a aparência, layout e design de uma página web, incluindo cores, fontes, espaçamento e muito mais. As suas regras consistem em seletores que identificam os elementos HTML e declarações que especificam como esses elementos devem ser estilizados.

JavaScript é uma linguagem de programação que permite adicionar interatividade e funcionalidade dinâmica às páginas web. Com ele é possível responder a eventos do usuário, validar formulários, fazer solicitações à API e muito mais. Ele é incorporado nas páginas HTML ou pode ser referenciado em arquivos externos.

Por exemplo, o código apresentado no anexo B cria uma página com título "Minha página Web", renderiza no corpo da página um título escrito "Bem-vindo ao meu site!" e um parágrafo contendo "Esta é uma página web simples."; o código css escrito dentro da *tag style* aumenta a fonte do título para 24 px e altera sua cor para azul, enquanto torna o parágrafo cinza e cria uma margem inferior de 20 px; por fim, o script JavaScript emite um alerta escrito "O título foi clicado!" caso o usuário clique no título da página.

2.6 Programação Web moderna

A programação web moderna com Svelte, TypeScript e Tailwind CSS representa uma abordagem contemporânea para o desenvolvimento de aplicações web, combinando várias tecnologias e práticas que visam eficiência, desempenho e produtividade.

Svelte é um *framework* de construção de interfaces de usuário que se diferencia de outras estruturas como React ou Angular. Em vez de ser executado no navegador, o Svelte compila o código em JavaScript puro durante a fase de compilação. Isso resulta em um código altamente otimizado e eficiente em tempo de execução. Ele se destaca por sua simplicidade e facilidade de aprendizado. Ele usa conceitos como componentes, reatividade e transições para criar interfaces de usuário dinâmicas. Além disso, o Svelte enfatiza o uso de sintaxe declarativa para manipular o DOM, o que leva a um código mais limpo e conciso.

TypeScript é uma linguagem de programação que estende o JavaScript, adicionando tipagem estática. Ele permite aos desenvolvedores definir tipos de dados para variáveis e parâmetros de função, tornando o código mais seguro e fácil de entender. Ele oferece autocompletar em ambientes de desenvolvimento integrado, melhor documentação de código e facilita a manutenção de grandes bases de código por pegar erros em tempo de compilação, melhorando a qualidade do código.

Tailwind CSS é uma estrutura de estilo CSS altamente configurável que segue a abordagem de "utility-first". Em vez de fornecer componentes predefinidos, o Tailwind oferece uma série de classes utilitárias que permitem estilizar elementos HTML diretamente no código HTML, o que simplifica a estilização de elementos da interface do usuário, economizando tempo e tornando o código CSS mais escalável e fácil de manter.

O código C é um exemplo básico de como essas 3 tecnologias trabalham em conjunto, onde lógica, estrutura de página e estilização coexistem num arquivo único e de fácil leitura. O código acima renderiza uma página com cor de fundo cinza claro com um título "Hello world!" sublinhado, com uma fonte grande e em negrito.

2.7 Conversor CC-CC

Conversores CC-CC (ou conversores DC-DC) Boost e SEPIC são dispositivos eletrônicos usados para aumentar a tensão de um sistema de corrente contínua (CC) para um nível desejado. Ambos são utilizados em diversas aplicações, como fontes de alimentação, sistemas de gerenciamento de bateria e conversores de energia.

2.7.1 Conversor Boost

O conversor Boost é projetado para elevar a tensão CC de entrada para uma tensão CC de saída maior. Ele é composto por três componentes principais: um interruptor (geralmente um transistor), um indutor e um diodo. O processo de funcionamento do conversor Boost é o seguinte:

1. Quando o interruptor (transistor) está fechado, a corrente flui do lado de entrada (V_{cc}) para o indutor, armazenando energia magnética nele.
2. Quando o interruptor é aberto, a energia armazenada no indutor é liberada. Isso causa um aumento na tensão no lado de saída.
3. O diodo evita que a corrente flua de volta para o lado de entrada, garantindo que a energia armazenada no indutor seja transferida para a carga.

O ganho de tensão do conversor Boost é determinado principalmente pela razão entre o tempo em que o interruptor está fechado e o tempo em que ele está aberto. A fórmula

básica para o ganho de tensão é $V_{out} = V_{in} / (1 - D)$, onde V_{out} é a tensão de saída, V_{in} é a tensão de entrada e D é a razão cíclica do interruptor (proporção do tempo em que o interruptor está fechado).

2.7.2 Conversor SEPIC

O conversor SEPIC (Single-Ended Primary Inductor Converter) é projetado para fornecer uma saída de tensão CC que pode ser maior ou menor do que a tensão de entrada. Ele é um conversor versátil que combina as funcionalidades de um conversor Boost e de um conversor Buck (que reduz a tensão). O processo de funcionamento do conversor SEPIC é o seguinte:

1. Quando o interruptor (geralmente um transistor) está fechado, a corrente flui do lado de entrada (V_{in}) para o indutor e a chave indutiva.
2. Quando o interruptor está aberto, a energia armazenada no indutor é transferida para a saída e para o capacitor.
3. O capacitor armazena a energia que será entregue à carga.

A principal vantagem do conversor SEPIC é a capacidade de manter uma tensão de saída regulada, independentemente de a tensão de entrada estar acima ou abaixo da tensão de saída desejada. Isso o torna útil em situações em que a tensão de entrada pode variar significativamente.

2.8 Rede de abastecimento de água

A rede de distribuição de água tem por objetivo colocar água potável à disposição dos consumidores, de forma contínua, em quantidade, qualidade e pressão adequadas, usando para isso uma série de tubulações e acessórios divididos em dois grupos: rede principal, conduto tronco ou canalização mestra, canalizações de maior diâmetro que abastecem as canalizações secundárias; e rede secundária, de menor diâmetro e cuja função é abastecer diretamente os pontos de consumo do sistema (SOBRINHO, 2003, p. 389-390).

Sobrinho (2003, p. 457) também destaca que existem perdas desde a captação até a entrega da água tratada ao consumidor, em grande parte causadas por operação e manutenção deficientes das tubulações e gestão inadequada das companhias de saneamento.

2.9 Sensores de vazão e pressão

Tanto os sensores de vazão quanto os sensores de pressão de água são fundamentais em aplicações que envolvem monitoramento, controle e automação de sistemas relacionados

à água. Os sensores de vazão são dispositivos usados para medir a taxa de fluxo de água em sistemas, como tubulações, dutos e canais, já os sensores de pressão de água são dispositivos projetados para medir a pressão da água em sistemas, como tanques, encanamentos e reservatórios.

Neste trabalho, foi utilizado o transdutor do tipo piezoresistivo exibido na Figura 3, ou seja, uma variação na sua resistência elétrica é causada pela deformação mecânica do elemento sensor. Dessa forma, ao aplicar uma diferença de potencial nos terminais do sensor é possível mensurar a pressão correspondente por meio da leitura da corrente que passa pelo circuito, como explicado pelo manual do anexo A, que também apresenta as especificações técnicas deste sensor, como o sinal de saída entre 4 e 20 mA.

Figura 3 – Sensor de pressão usado.



Fonte: o autor

Na Figura 4, está ilustrado o sensor de fluxo (fluxímetro) usado no projeto, o qual possui um rotor de água e um sensor de efeito Hall. Quando a água flui através do rotor, faz com que este gire, e as mudanças de velocidade que variam de acordo com o fluxo de água são lidas pelo sensor de efeito hall, que emite o pulso de sinal correspondente. Ou seja, este não é um sensor pensado para medir consumo, mas sim a vazão instantânea.

Figura 4 – Sensor de vazão usado



Fonte: o autor

3 Materiais e Métodos

Nesta seção serão apresentados os materiais utilizados na realização do projeto. Uma parte dos experimentos foi feita no Laboratório de Eficiência Energética e Hidráulica em Saneamento da Universidade Federal da Paraíba (LENHS/UFPB), onde são utilizados conjuntos motor-bomba (CMB), dutos de 50 mm para a canalização da água e reservatórios para o armazenamento da água (Figura 5). Os ensaios feitos no laboratório tiveram como objetivo analisar coleta de dados de pressão e vazão em uma das saídas de água da planta principal por meio da conexão do microcontrolador a um sensor de pressão e a um sensor de vazão devidamente instalados.

Figura 5 – Laboratório de Eficiência Energética e Hidráulica em Saneamento (LENHS).



Fonte: O autor.

3.1 Materiais Utilizados

Para a criação do *hardware* desenvolvido neste projeto foram utilizados os seguintes componentes:

- 1 microcontrolador ESP32;
- 1 bateria 16340 e suporte;
- 1 módulo conversor analógico digital ADC ADS1115 16 Bit i2C;
- 2 capacitores eletrolíticos de $100\mu F$;
- 2 capacitores eletrolíticos de $10\mu F$;
- 1 diodo Schottky 1N5819;
- 2 indutores de $470\mu H$;
- 4 resistores de $100k\Omega$;
- 1 resistor de $1k\Omega$;
- 1 resistor de 100Ω ;
- 1 regulador linear LDO HT7333-A;
- 1 transistor Mosfet IRF4905;
- 3 transistores NPN BC548;
- 1 transistor PNP 2N5401;
- fios para conexão dos componentes;
- placa de prototipagem;

Para os testes foram utilizado os seguintes materiais:

- 1 transdutor de pressão Druck PTX 7217, cujo manual está inserido no anexo A, escolhido por questão de compatibilidade;
- 1 sensor de vazão YF-S201;
- 1 osciloscópio DSOX2012A, do LEAD;
- 1 multímetro digital Minipa ET-1002;

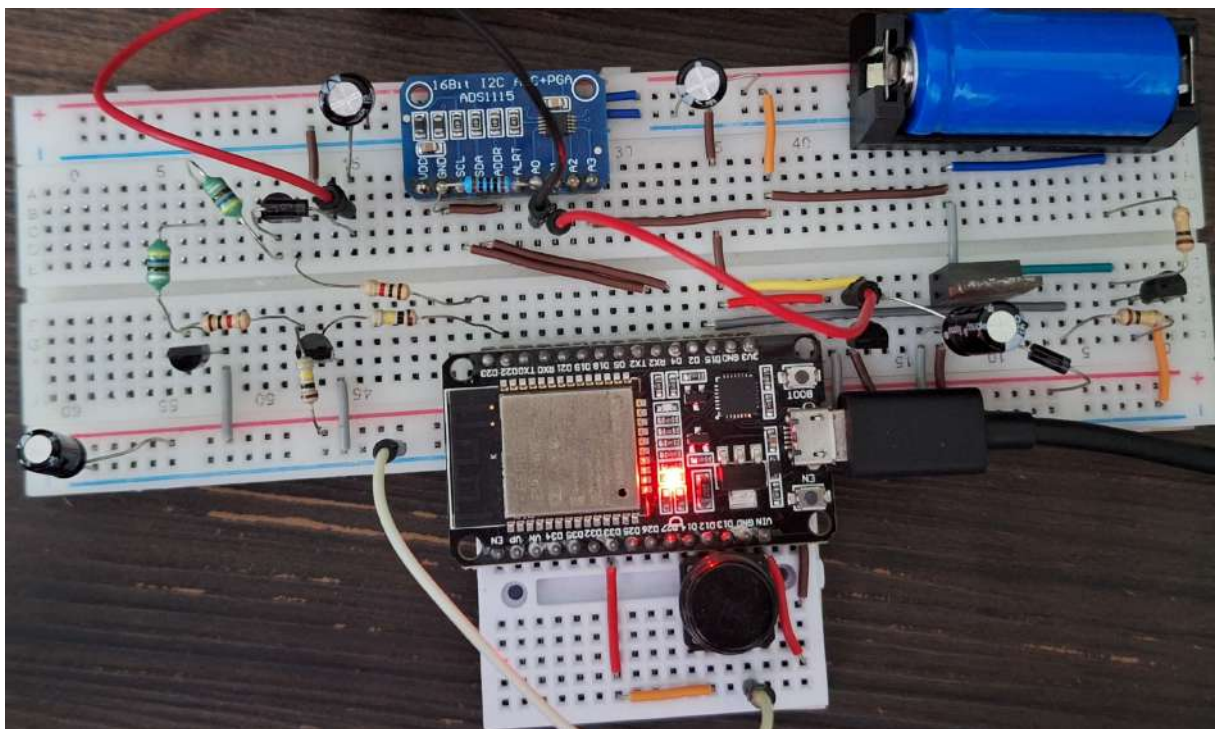
Por fim, para a realização de alguns experimentos, foi usado uma chave tátil de 4 terminais para simular pulsos de um sensor de consumo.

3.2 Desenvolvimento do hardware

Foi desenvolvido um protótipo funcional para o monitoramento de grandezas elétricas e hidráulicas. Ele gerencia o processo de carregamento da bateria, registra os pulsos do sensor de vazão, realiza a conversão de tensão e fornece energia ao sensor de pressão somente quando necessário. Além disso, ele registra o valor da pressão atual por meio de sua saída de corrente e, por fim, transmite todas essas informações via comunicação sem fio (WiFi) para o servidor, como será detalhado mais adiante.

Na Figura 6, ilustram-se as conexões elétricas do protótipo desenvolvido usadas para os testes relacionados à autonomia e ao registro de pulsos com base nos hidrômetros utilizados pela CAGEPA (Companhia de Água e Esgotos da Paraíba). Durante os testes, os registros eram acionados por meio do pressionamento de um botão tátil, com uma frequência máxima de 1 Hz, que corresponde à frequência máxima do medidor, conforme explicado no manual (SAGA, 2022).

Figura 6 – Hardware desenvolvido



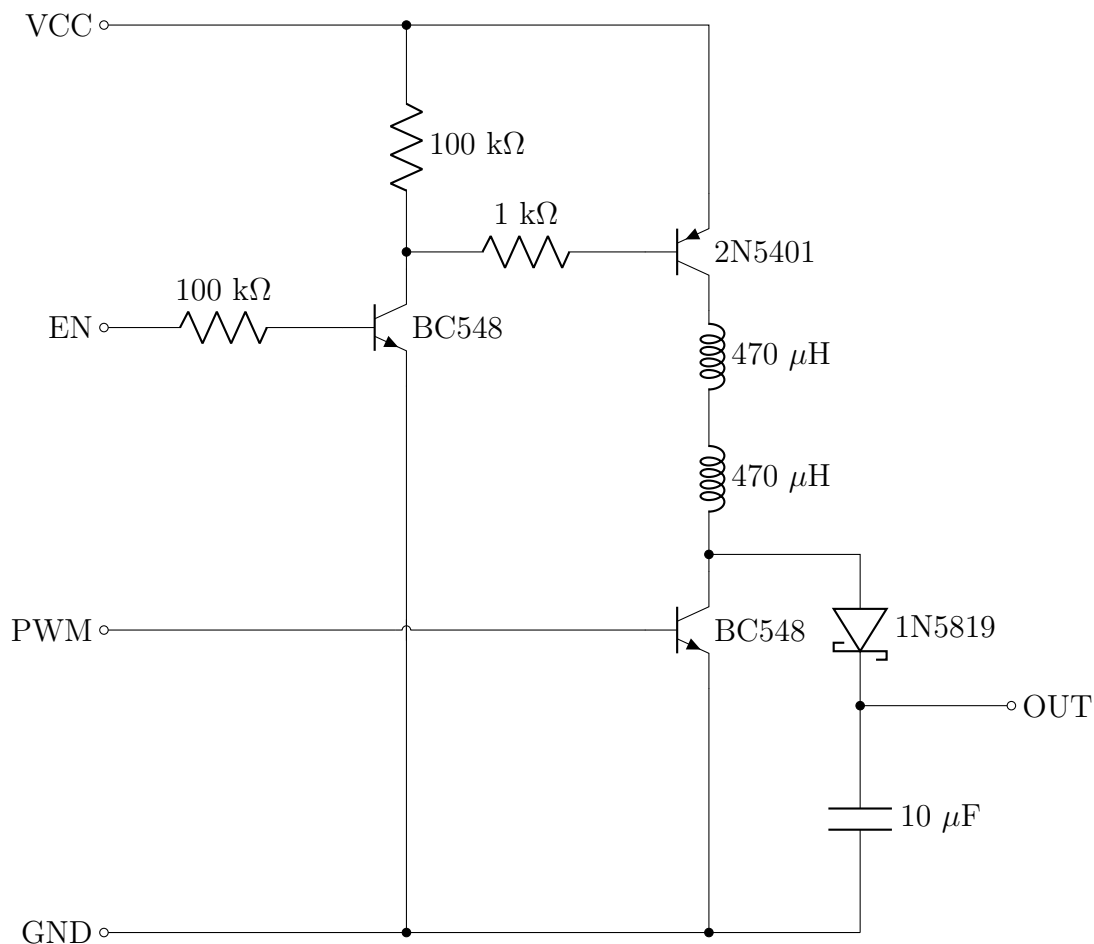
Fonte: O autor

Para tornar a compreensão mais acessível ao leitor, esta seção foi dividida em três partes distintas: Na primeira, é explicado o funcionamento do conversor *boost* e como ele gera a tensão necessária para o sensor de pressão funcionar adequadamente. Na segunda parte, é explicado o circuito da bateria e da gestão de energia. Por fim, na terceira parte, é mostrado como os componentes das duas primeiras partes se integram com o conversor ADC e o ESP32.

3.2.1 Elevador de tensão

O diagrama esquemático ilustrado na Figura 7 corresponde ao circuito elevador de tensão desenvolvido para fornecer a tensão adequada para o funcionamento do sensor de pressão, que opera a 12V, a partir da bateria de 3.7V que está em uso. Além da entrada de sinal PWM convencional, usada para conectar os indutores ao potencial terra, o circuito também possui uma entrada EN, que pode ser controlado pela saída digital do ESP graças ao transistor NPN, que faz a conversão de tensão necessária com o VCC e assim permite a passagem de corrente quando ativada e corta essa alimentação quando em nível baixo, uma característica particularmente útil para prolongar a autonomia da bateria, já que sem ela, mesmo que o terminal do PWM permanecesse em nível baixo, ainda haveria passagem de corrente para a o pino de saída OUT , que é por onde é fornecida a tensão necessária para alimentar o sensor de pressão.

Figura 7 – Diagrama do circuito elevador de tensão



Fonte: O autor

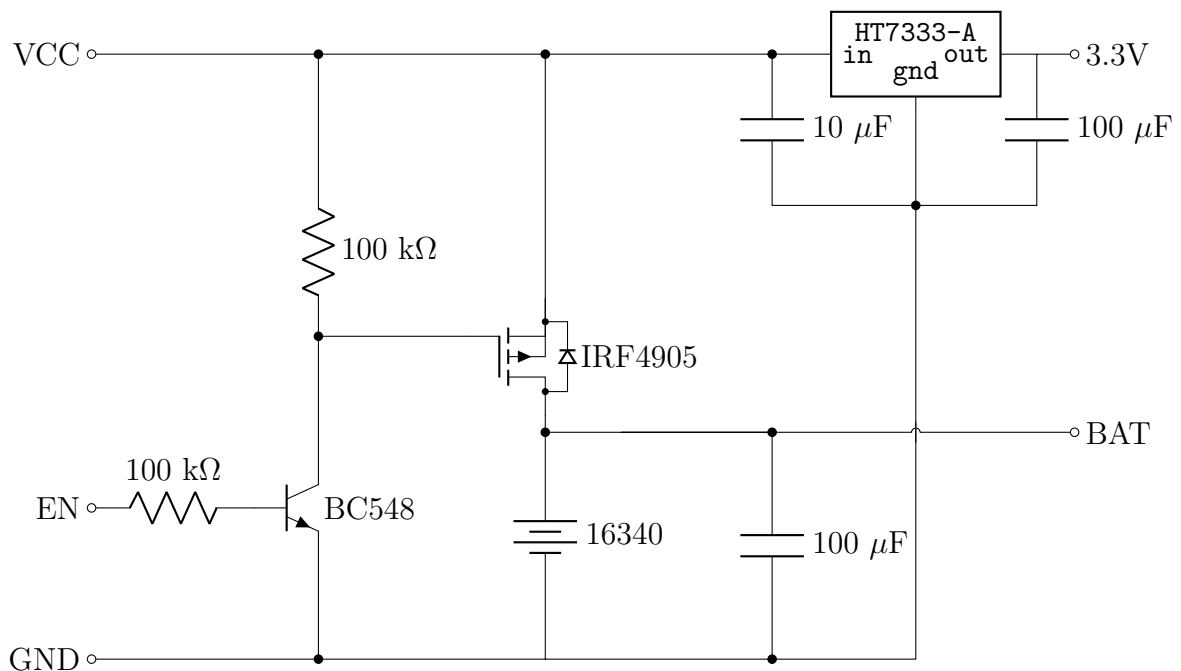
Uma outra alternativa interessante seria ter usado um conversor SEPIC, assim não seria necessário cortar a alimentação do circuito usando o transistor PNP. Frequências mais altas de comutação também poderiam ter sido aplicadas, já que o ESP consegue

gerar sinais de até 20 MHz, a fim de reduzir as indutâncias usadas, o que reduziria a resistência associada a elas e assim elevaria a eficiência do circuito. Outra coisa que também poderia ter sido implementada a fim de acelerar a estabilização da tensão, seria cortar a corrente de saída usando um transistor, o que faria com que capacitor fosse carregado mais rapidamente.

3.2.2 Bateria e gestão de energia

Como mostrado na Figura 6, a bateria foi posicionada num par de trilhas laterais da placa de prototipagem, enquanto o terminal positivo do par de trilhas do lado oposto da placa está identificado como VCC no diagrama ilustrado na Figura 8, pino este conectado ao pino V_{in} do ESP, sendo assim energizado pelo USB quando o ESP está conectado a alguma fonte de energia externa, como o computador, e que pode ser usado para a conexão de qualquer fonte de energia com tensão máxima de 5 V, como um mini gerador hidrelétrico ou um painel solar. Ele também está conectado à entrada do regulador LDO, como pode ser visto no diagrama da Figura 8, responsável por abastecer o ESP com uma tensão estabilizada de 3.3 V sem para isso desperdiçar muita energia, já que ele é um regulador linear de tensão do tipo LDO (low-dropout), ou seja, a queda de tensão entre seus pinos de entrada e saída é inferior a 100 mV, 13 vezes menos do que a queda de tensão do regulador linear padrão do ESP32, que é o AMS1117, cuja queda de tensão é de aproximadamente 1.3 V.

Figura 8 – Diagrama do circuito de energia



Fonte: O autor

Por fim, como pode ser visto na Figura 8, a bateria é conectada ao VCC por meio

do MOSFET, que permite a passagem de corrente para a bateria quando o pino *EN* está em nível alto e libera a passagem em sentido reverso quando a tensão da bateria é maior do que a de VCC graças ao seu diodo. Como a tensão da bateria também precisa ser lida, foi adicionada a saída *BAT* ao diagrama para mostrar que ela é usada por um circuito externo, no caso, o módulo conversor analógico-digital que será explicado em sequência.

Não foi adicionado um sistema de controle de corrente de carregamento porque durante os testes, mesmo usando uma fonte de bancada de 5V para carregar a bateria totalmente descarregada, a corrente de carregamento nunca ultrapassou 200 mA, valor totalmente dentro do limite aceitável.

3.2.3 Integração com ADC e ESP

Para a integração dos circuitos mencionados anteriormente, inicialmente, foi estabelecida a conexão do VCC entre ambos os circuitos. Além disso, o pino VDC do ADC foi conectado diretamente à bateria para permitir a leitura de sua tensão. Para manter a consistência elétrica, o GND de todos os elementos foi conectado entre si.

No que diz respeito ao funcionamento do conversor ADC, o pino ADDR foi ajustado para utilizar o endereço I2C padrão por meio da sua conexão ao pino GND. Sua tensão de entrada pode variar de 2 V a 5 V, seu consumo de corrente é inferior a 1 mA e suas leituras tem 16 bits de precisão, 4 bits a mais do que o conversor ADC interno do ESP e com a vantagem de suas leituras não serem ruidosas, mas sim precisas, como é comum ocorrer em módulos dedicados para uma função específica. Quanto aos pinos SDA e SCL, eles foram conectados respectivamente aos GPIO 4 e 16 do ESP.

Em relação às entradas analógicas do conversor ADC, as conexões foram efetuadas conforme segue: A0 foi conectado a um resistor de 100Ω entre o terminal negativo do sensor de pressão e o terra; o A1 foi associado ao V_{cc} ; o A3 foi conectado diretamente à bateria; e o A2 não foi conectado a nada pois o plano original era fazer o monitoramento da tensão gerada pela microturbina com ele, porém a sua utilização foi comprometida devido a problemas ocorridos durante os experimentos.

As demais conexões foram feitas da seguinte forma:

- *EN* do carregador: responsável por permitir a passagem de corrente para a bateria quando encontra-se em nível lógico alto, foi conectado ao GPIO 15;
- *EN* do conversor elevador de tensão: permite a passagem de corrente para o circuito elevador de tensão quando em nível lógico alto, conectado ao GPIO 22;
- *PWM* do *boost*: responsável pela elevação da tensão, conectado ao GPIO 23;
- registro de pulsos é feito pelo GPIO 12.

3.3 Desenvolvimento do *software* embarcado

Esta seção apresenta uma visão geral do desenvolvimento do código em Rust e explicadas as razões subjacentes às decisões tomadas. É importante observar que para compreender completamente o código, é necessário ter conhecimento considerável da linguagem, e aqueles familiarizados com Rust podem acessá-lo diretamente no Anexo E.

Primeiro, são abordadas a preparação do ambiente de desenvolvimento, destacando as instalações e configurações essenciais que possibilitaram a programação do ESP com Rust; a criação do projeto e sua estrutura de arquivos, oferecendo uma breve descrição de cada componente; quais as bibliotecas, variáveis de ambiente e constantes foram usadas neste projeto; e um diagrama geral com uma explicação superficial sobre os trechos considerados particularmente relevantes.

3.3.1 Preparação de ambiente

Todo o processo de preparação seguido está descrito na seção *Setting Up a Development Environment* do manual da ESPRESSIF (2023). Em resumo, a forma mais simples de ter preparar o ambiente consiste em primeiro instalar o Rust por meio do *rustup*, seu instalador oficial e disponível para download no Windows, após isso basta executar o comando *cargo install espup* seguido pelo comando *espup instal* para que todas as ferramentas necessárias sejam automaticamente instaladas no computador. Em relação à interface de desenvolvimento, a recomendada é o Visual Studio Code por possuir uma extensão chamada *rust-analyser*, que simplifica todo o processo de desenvolvimento.

3.3.2 Criação de projeto e estrutura de arquivos

Para facilitar a criação do projeto, foi usado um modelo fornecido pela ESPRESSIF gerado pelo comando *cargo generate esp-rs/esp-template*, que é executado por meio da ferramenta *cargo-generate*, instalável pelo comando *cargo install cargo-generate*. Durante a criação do projeto, a ferramenta faz diversas perguntas ao usuário para que ao final o código já possa ser testado no ESP escolhido. Na Figura 9 é exibida a saída preenchida de geração de um projeto com as mesmas configurações usadas para geração do código descrito nesta seção e a Figura 10 apresenta a estrutura de arquivos inicial do projeto, gerada automaticamente.

Começando pelo arquivo *config.toml* da pasta *.cargo*, adicionado ao Anexo D, onde define-se qual o programa responsável por executar o executável gerado após a etapa de compilação por meio da variável *runner*; em *build* são definidas algumas *flags* para o compilador e qual o *target* do projeto; e o *build-std*, responsável por compilar todas bibliotecas do Rust do zero ao invés de usar arquivos pré-compilados.

Figura 9 – Saída da geração de projeto

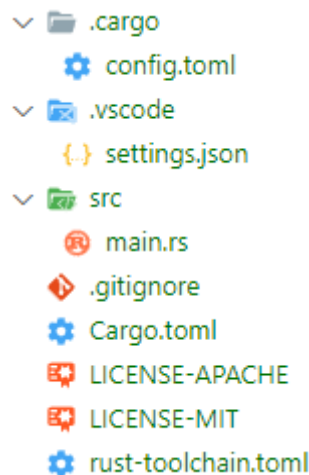
```

project-name: default-template ...
Generating template ...
✓ Which MCU to target? · esp32
✓ Configure advanced template options? · true
✓ Enable allocations via the esp-alloc crate? · false
✓ Configure project to use Dev Containers (VS Code and GitHub Codespaces)? · false
✓ Configure project to support Wokwi simulation with Wokwi VS Code extension? · false
✓ Add CI files for GitHub Action? · false
✓ Setup logging using the log crate? · false
✓ Enable WiFi/Bluetooth/ESP-NOW via the esp-wifi crate? · true

```

Fonte: O autor

Figura 10 – Estrutura inicial de arquivos do projeto



```

└─ .cargo
   └─ config.toml
└─ .vscode
   └─ settings.json
└─ src
   └─ main.rs
└─ .gitignore
└─ Cargo.toml
└─ LICENSE-APACHE
└─ LICENSE-MIT
└─ rust-toolchain.toml

```

Fonte: O autor

O arquivo *settings.json* contém apenas uma linha escrita *"rust-analyzer.check.allTargets": false*, que informa à extensão *rust-analyzer* para apenas fazer a checagem de código no *target* definido no arquivo *config.toml*, evitando assim erros ao tentar compilar o código para plataformas não suportadas. *rust-toolchain.toml* também é um arquivo simples, pois sua única função é definir qual "versão" do Rust deve ser usada, o que é feito em apenas duas linhas: *[toolchain]* e *channel = "esp"*, que informa ao Cargo de que deve ser usada a versão da ESPRESSIF para compilar este projeto. O arquivo *.gitignore* gerado detalha algumas pastas e arquivos temporários que devem ser ignorados pelo Git, como a pasta *target*, que armazena centenas de arquivos temporários durante a compilação.

Por fim, os dois arquivos principais de qualquer projeto Rust são o: *Cargo.toml*, onde são definidas as informações de versão, nome, edição do Rust, autores e bibliotecas usadas; e o arquivo *main.rs*, onde a programação propriamente dita é escrita.

3.3.3 Bibliotecas utilizadas

Aqui serão listadas todas as bibliotecas utilizadas e uma breve descrição de qual a sua utilidade:

- `esp32-hal`: camada de abstração do *hardware*, é base de todo o projeto e fornece acesso a todos os periféricos básicos do ESP32;
- `esp-backtrace`: fornece tratamento às exceções graves geradas com o *panic*, ou seja, aquelas onde o programa é prontamente interrompido, e exibe todas as informações importantes no erro no Serial;
- `esp-println`: funções de escrita no serial *print*, *println* e *dbg* que seguem o padrão da linguagem;
- `embedded-storage`: "contratos" usados como base para implementar a camada de acesso à memória de microcontroladores;
- `esp-storage`: camada de acesso à memória Flash do ESP, permitindo acesso fácil à leitura e escrita de informações;
- `embedded-svg`: "contratos" usados como base para implementar acesso a WiFi e bluetooth em microcontroladores com suporte;
- `esp-wifi`: driver para acesso WiFi, Bluetooth e ESP-NOW das placas da ESPRESSIF;
- `smoltcp`: *stack* TCP/IP desenhada para aplicações embarcadas com Rust;
- `nb`: camada mínima e reusável de I/O não-bloqueante;
- `mqtttrs`: utilitários de codificação e decodificação de pacotes MQTT;
- `dotenvy_macro`: carregamento de variáveis escritas em arquivo `.env` em tempo de compilação;
- `serde`: framework genérico para serialização e desserialização de dados, é um dos pacotes mais populares da comunidade e já teve mais de 220 milhões de downloads;
- `serde-json-core`: implementação da serialização e desserialização de JSON em projetos embarcados sem necessitar de alocação dinâmica de memória;
- `ads1x1x`: driver para os conversores analógico para digital de baixo consumo ADS1013, ADS1014, ADS1015, ADS1113, ADS1114 e ADS1115;
- `desse`: serialização e desserialização de dados eficiente para tipos com tamanho conhecido em tempo de compilação;

3.3.4 Variáveis de ambiente e constantes

Ao todo, há quatro valores que regem o comportamento do programa, são eles:

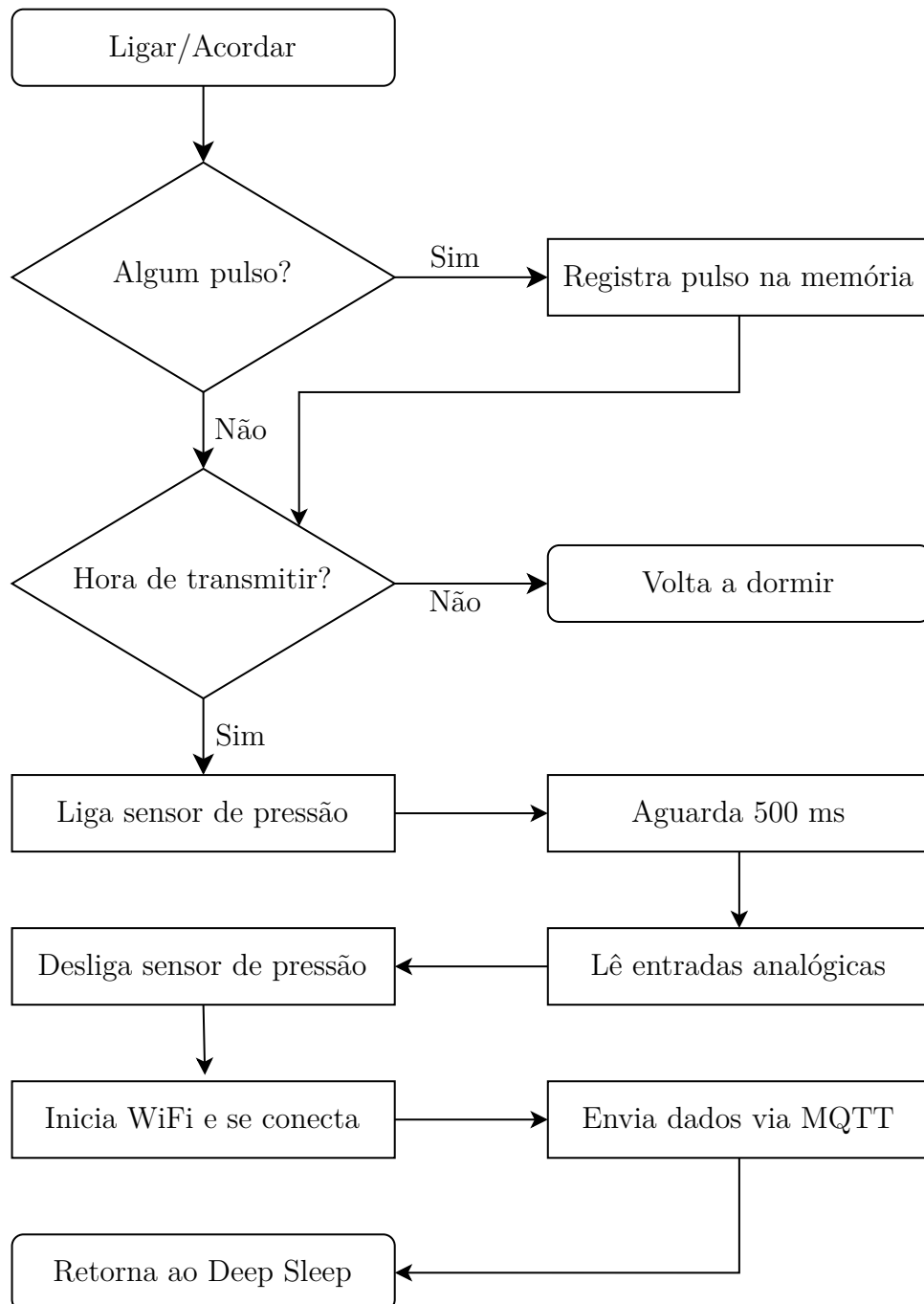
- *FLASH_ADDR*: local onde inicia a seção da memória *flash* responsável por armazenar informações, por padrão ela tem um offset de 36.864 bytes, ou, 9000 em hexadecimal, como informado pela documentação oficial;
- *INTERVAL_SEC*: intervalo em segundos entre cada transmissão de estado do ESP para o servidor;
- *SSID* e *PASSWORD*: respectivamente nome e senha da rede WiFi que será usada, configurados dentro de arquivo *.env*, ignorado pelo Git para evitar vazamento de informação sigilosa, e importados durante a compilação por meio do macro *dotenv!*.

Enquanto as duas primeiras constantes podem ser definidas no código sem problema, já que não são sigilosas, as outras duas são mantidas de fora para que ninguém descubra a senha do WiFi por meio do repositório Git, o que é apenas um dado simples que não teria consequências graves caso vazasse, porém o mesmo não pode ser dado de chaves de uma API interna de uma empresa, por exemplo, e que vazam diariamente porque muitos programadores iniciantes (principalmente aqueles que não são especificamente da área, como engenheiros) não sabem usar arquivos de variáveis de ambiente (*.env*) para separar as informações confidenciais do restante do código, o que também ocorre porque poucas linguagens fazem a importação dos dados deste tipo de arquivo de forma tão simples quanto o Rust, que usa um simples macro para fazer isso e verificar se o dado está informado corretamente em tempo de compilação.

3.3.5 Diagrama geral

O sistema pode ser resumido pelo diagrama da Figura 11, a qual apresenta superficialmente todas as etapas que devem ser executadas pelo ESP para que ele possa regularmente enviar os dados para o servidor com uma autonomia razoável de bateria, já que o sensor de pressão só é ativado pelo tempo mínimo necessário para que a leitura seja feita e ele é desligado tão logo ela é realizada e o ESP permanece em modo *sleep* na maior parte do tempo.

Figura 11 – Diagrama geral do sistema.



Fonte: O autor.

3.4 Desenvolvimento do servidor

Como o foco deste projeto foi o desenvolvido do *software* e *hardware* embarcado, o servidor desenvolvido foi bem simples e tinha como única tarefa servir de *broker* MQTT, registrar todas as mensagens recebidas em um arquivo *result.json* e disponibilizar este arquivo para download a fim de utilizá-lo para posterior coleta de resultados.

Para o *deploy*, foi usada uma estratégia mais conservadora, que é uma máquina virtual gratuita na AWS EC2, mais exatamente uma *t2.micro*, com apenas um núcleo virtual, um GiB de RAM e 8 GB de memória gp2 (SSD de uso geral), ou seja, um ambiente bem limitado, o que não é problema para o Rust.

3.4.1 Criação e bibliotecas

Para a criação do projeto foi usado o comando padrão *cargo init*, que inicializa o projeto dentro da pasta vazia selecionada. Feito isso, foram adicionadas as bibliotecas:

- *axum*: um dos principais frameworks web disponíveis para Rust e com mais de 23 milhões de downloads, usado para disponibilizar o arquivo de resultado para download;
- *rumqtt*: broker MQTT de alta performance totalmente escrito em Rust e bem personalizável;
- *config*: leitura de arquivo de configurações em tempo de execução, utilizado pelo *rumqtt* para definir as configurações do broker;
- *tokio*: runtime assíncrono que permite a execução de diversos processos ao mesmo tempo mesmo em máquina de um núcleo a um baixo custo;
- *tower-http*: funções úteis para um servidor HTTP, usado para fornecer o arquivo *result.json* para download.

3.4.2 Código desenvolvido

O código do servidor desenvolvido pode ser lido no anexo F e aqui será apresentada uma breve descrição da função de cada uma das 3 funções.

A função *serve_dir* é responsável por permitir o download do arquivo de resultados por meio da função *ServeDir* fornecida pela biblioteca *tower-http*, que foi configurada para permitir o download de qualquer arquivo do pasta raiz do programa via HTTP por meio da porta 8080, que é redirecionada para a porta 80 por meio do comando *iptables -t nat -A PREROUTING -p tcp -dport 80 -j REDIRECT --to-port 8080*, executado por permissões de administrador no sistema operacional Ubuntu.

A razão de ser necessário esse redirecionamento de porta é que a porta 80 é protegida e apenas programas com permissão de administrador podem acessá-la, o que jamais deve ser o caso de uma aplicação que lida com tráfego externo da rede, pois isso abre inúmeras brechas de segurança. Por isso, o código do servidor sempre deve ser executado sem essas permissões e todas as portas por ele usadas devem ser redirecionadas por outro meio qualquer, e o escolhido neste trabalho foi o mais simples: o *iptables*.

Em relação ao registro das informações recebidas pelo *broker* MQTT, essa tarefa é feita pela função *process_msg*, que tem apenas 3 parâmetros, uma entrada de dados do broker, o arquivo para onde os dados são enviados e um booleano que indica se aquela mensagem é a primeira a ser registrada no arquivo, o que é útil para a sua correta formatação. Ele aguarda novas mensagens serem recebidas bloqueando a thread principal e, caso a mensagem seja de fato do tipo *publish*, ele registra o corpo da mensagem no arquivo e salva-o.

Por fim, a entrada da aplicação é chamada de *main* e ela é do tipo assíncrono graças à biblioteca *tokio*, que é a forma mais popular de trabalhar com tarefas assíncronas em Rust. Logo no começo ela inicia a função *serve_dir* para que o arquivo de resultados possa ser baixado, então lê o arquivo de configurações do broker e usa-o para iniciá-lo, se inscreve em todos os tópicos por meio do caracter especial “#” e inicia a tarefa principal do broker em outra thread. Por fim, o arquivo de resultados é gerado e um loop de processamento de mensagens é iniciado, ativo até que a aplicação seja encerrada.

3.5 Desenvolvimento do aplicativo de monitoramento

A Figura 12 ilustra a estrutura de arquivos do projeto do aplicativo, onde nota-se que a quantidade de arquivos de configuração num projeto Web moderno é relativamente grande se comparado ao que foi visto nos outros dois projetos apresentados anteriormente. A razão disso é que para a criação de qualquer aplicativo, por mais simples que seja, é necessário o uso de diversas ferramentas diferentes e cada uma tem seu próprio arquivo de configurações. O motivo dessa grande quantidade de arquivos não ser necessariamente um problema é que todos eles tem ótimas documentações explicando o que cada configuração faz e, no geral, quase nunca é necessário alterar tais valores.

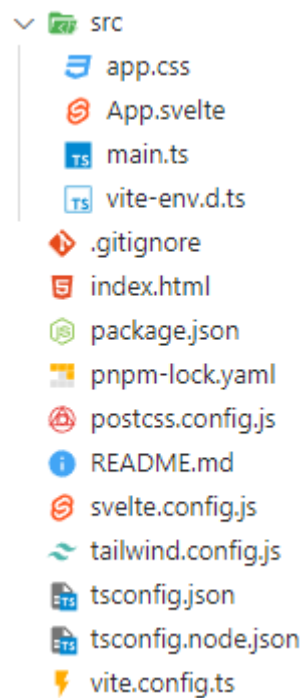
Para a geração do projeto foi usado o template do Vite para criação de projetos com Svelte e TypeScript, usado por meio do comando `pnpm create vite my-vue-app --template svelte-ts`, conforme descrito no guia de início rápido do Vite (2023). Para a configuração da biblioteca CSS foi usado o passo a passo do guia de instalação em projetos Svelte do CSS (2023).

As duas bibliotecas usadas no projeto foram: *chartist* para geração de gráficos e *mqtt* para conexão com o broker via websocket e assim, combinando o poder de ambas

com as facilidades do Svelte, foi possível renderizar no aplicativo Web as informações enviadas pelo ESP em tempo real usando apenas o código apresentado no Anexo E.

Resumidamente, o código começa importando as bibliotecas necessárias; define na interface qual a estrutura dos dados recebidos pelo MQTT; armazena os últimos valores recebidos e todos os 4 gráficos que são iniciados logo após a página ser renderizada por meio da função `onMount`; conecta-se ao broker hospedado na AWS, se inscreve em todos os tópicos após a conexão ser estabelecida e registra a função que será executada a cada mensagem executada, que filtra apenas as últimas 100 mensagens e atualiza as informações de cada um dos gráficos. Em relação ao HTML, a renderização de sua forma final pode ser vista na figura 13.

Figura 12 – Estrutura de arquivos do aplicativo Web de monitoramento



Fonte: o autor

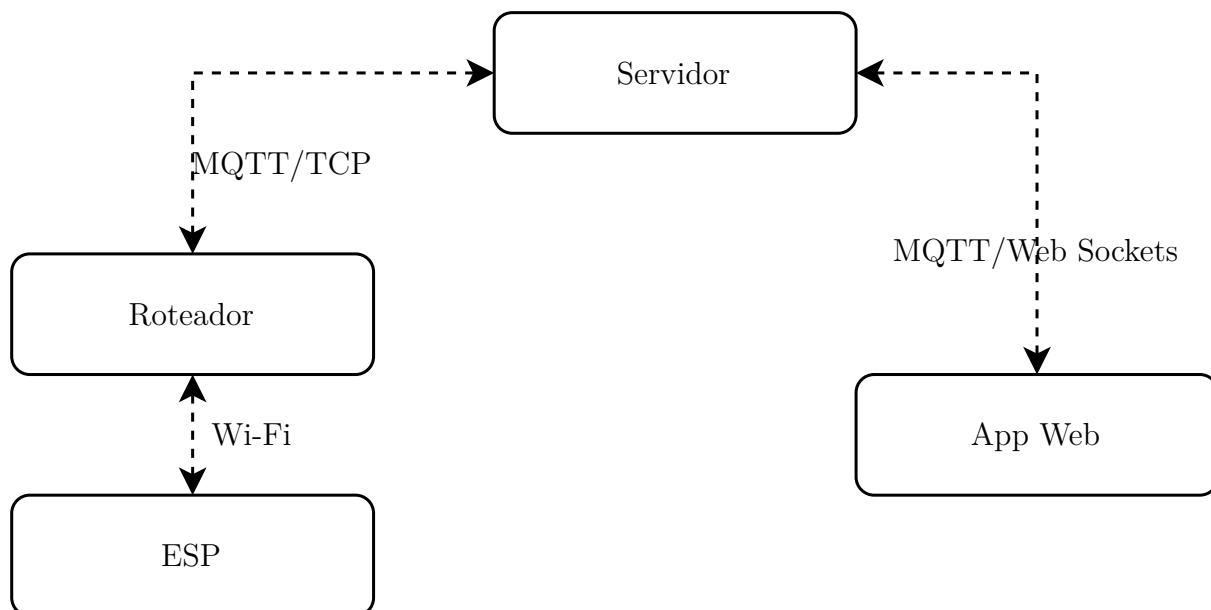
Figura 13 – Captura de tela do aplicativo de monitoramento



Fonte: O autor

3.6 Comunicação

Figura 14 – Diagrama simplificado de comunicação.



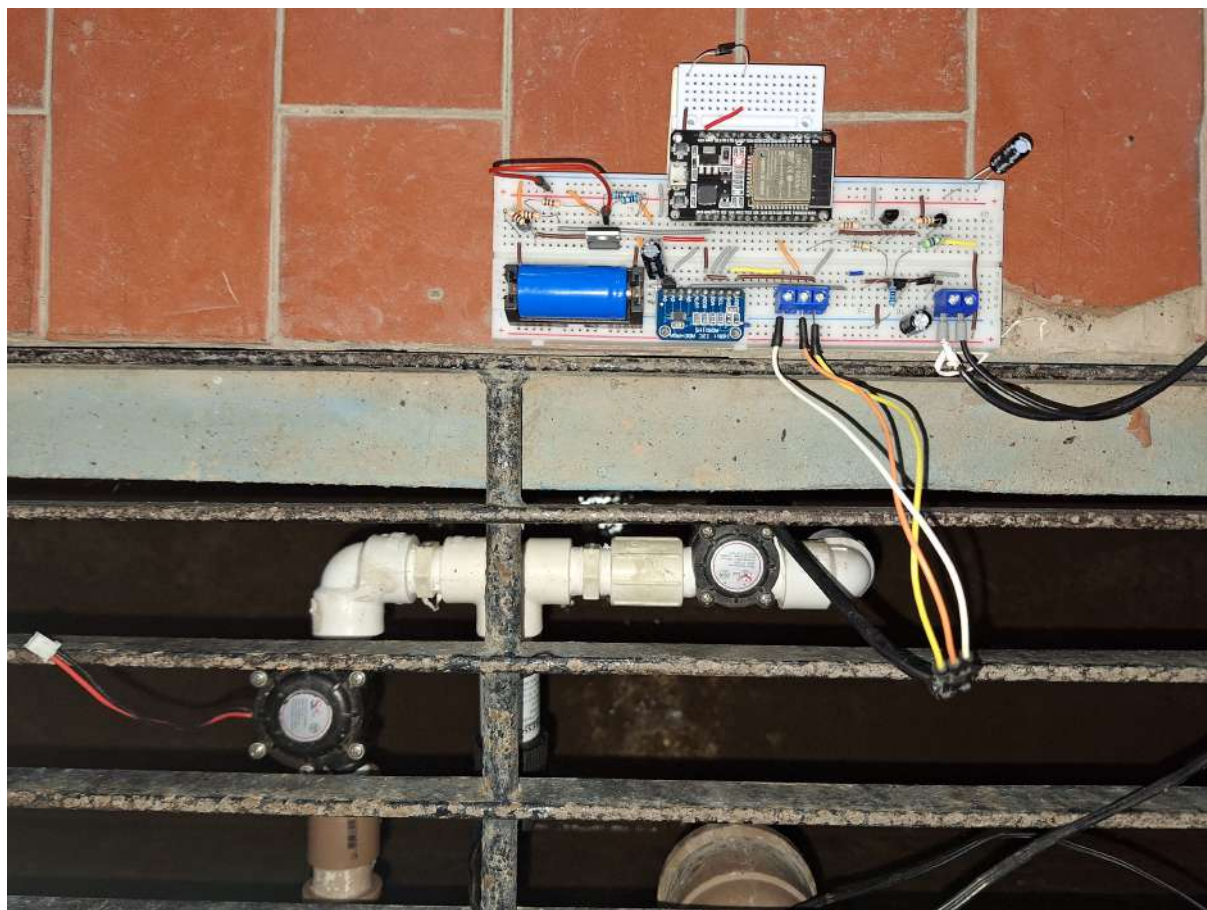
Fonte: o autor.

Para a comunicação entre o ESP e o servidor e entre o aplicativo Web e o servidor não foi implementada nenhuma camada de criptografia e autenticação, etapa essencial em um projeto comercial.

3.7 Monitoramento contínuo de pressão e vazão

Para o teste de monitoramento contínuo de pressão e vazão foram usadas as instalações do LENHS, que disponibiliza, em uma de suas saídas de água, um conjunto de mini gerador hidráulico (inoperante no momento dos testes), um sensor de pressão e um sensor de vazão, como mostra a Figura 15.

Figura 15 – Instalação do protótipo de monitoramento no LENHS



Fonte: o autor

Como apresentado na Figura 5, o laboratório onde foram feitos os testes é grande e o local onde foi feita a instalação é distante da sala de controle, onde está instalado o roteador de Internet, por isso foi necessário usar um repetidor de sinal para permitir que o ESP pudesse se conectar à internet, como mostrado na Figura 16, onde o repetidor está instalado no canto superior direito da foto, enquanto o *hardware* do projeto foi instalado no canto inferior esquerdo da foto, o que representa uma distância de aproximadamente 6 metros.

Feitas as ligações, foi feito o controle da frequência de rotação de uma bomba para conseguir simular diferentes vazões e pressões nesta saída de água e assim conseguir descobrir como é o comportamento do sistema sob as variações dessas grandezas. Para

Figura 16 – Local de instalação do hardware do projeto e do repetidor



Fonte: o autor

isso foram feitos dois ensaios. No primeiro a frequência da bomba foi alterada entre 30 e 50 Hz em degraus de 5 e 10 Hz enquanto o ESP enviou um registro por segundo para o broker; já no segundo, a frequência foi aumentada gradativamente de 30 Hz até chegar a 50 Hz com degraus de apenas 1 Hz em um intervalo regular de 30 segundos entre cada alteração enquanto o ESP capturou dados a cada 100 ms e os transmitiu em lotes de 10 a cada um segundo.

Como a frequência de captura e transmissão de dados destes testes foi muito alta, não seria possível usar o mesmo código usado nas análises de autonomia de bateria e por isso foi necessário desativar o sistema de *sleep*, manter o ESP ativo e conectado ao WiFi constantemente e usar um loop para registrar e transmitir os dados com um intervalo regular entre cada ação. Com estas medidas, é de se esperar uma severa diminuição da autonomia, mas como os tempos de cada ensaio são inferiores a 10 minutos, já que, por exemplo, o segundo leva pouco mais de 10 minutos para ser executado, a autonomia da bateria não foi uma preocupação durante a sua execução.

Por fim, para ter certeza de que a coleta estava ocorrendo corretamente, ao lado do painel de controle da bomba do laboratório feito em LabVIEW, foi mantido aberto o aplicativo web de monitoramento para que as variáveis do teste pudessem ser acompanhadas junto com as variáveis de energia do sistema, como a tensão de entrada do ESP e a tensão da bateria. Então, ao final de cada ensaio, foi feito o download do arquivo JSON gerado no servidor para posterior análise de resultados, que serão apresentados no próximo capítulo.

3.8 Análise do sensor de pressão com conversor *boost*

Esta análise foi feita usando as instalações do LEAD (Laboratório de Eletrônica Analógica e Digital) da UFPB, mais especificamente, usando um de seus osciloscópios DSOX2012A para avaliar as tensões relacionadas ao conversor boost para com isso entender como é o comportamento do sensor de pressão ao ser alimentado com este conversor por uma bateria e assim obter métricas importantes, como quanto tempo ele precisa permanecer ativo para que uma medição precisa da pressão possa ser lida pelo módulo conversor analógico para digital.

3.9 Análise de autonomia de bateria

Para a análise de autonomia foi necessário carregar e descarregar a bateria várias vezes até coletar dados suficientes para o treinamento da rede neural detalhado na seção subsequente. Durante os testes, o ESP enviou a cada um minuto: uma leitura da tensão no resistor conectado ao sensor de pressão para que houvesse a certeza de que ele ainda estava operacional durante toda a execução do teste; a tensão da bateria e da entrada do conversor LDO para poder treinar a IA e para poder saber qual a queda de tensão sobre o transistor MOSFET.

Uma das grandes dificuldades deste experimento foi que, ao aumentar a autonomia, é necessário aumentar o intervalo entre cada registro e transmissão de dados, porém ao fazer isso tanto se prolonga o tempo de testes quanto diminui a quantidade de pontos registrados ao longo do experimento, sendo necessário sempre buscar um equilíbrio razoável entre a autonomia de bateria e a quantidade de dados coletados a cada ensaio.

Ao final de todos os testes, o arquivo JSON com todos os dados coletados foi baixado para que pudesse ter início a última parte experimental do trabalho, detalhada na próxima seção.

Uma forma mais eficiente para estimação de autonomia seria criar um gêmeo digital para o sistema e usá-lo para estimar o comportamento do sistema ao longo do tempo sob diferentes parâmetros do sistema, que teria um grande aumento de autonomia caso fosse adotada alguma estratégia de *power harvesting*, como a adição de uma micro turbina hidráulica, e o registro de pulsos fosse feito usando um ATTiny para que o ESP não precisasse ser acionado a cada pulso do sensor de consumo. Testes com baterias de melhor autonomia também seriam interessantes, já que algumas, como a LG MJ1, tem capacidade de até 3500 mAh.

3.10 Estimação de carga de bateria em uso com IA

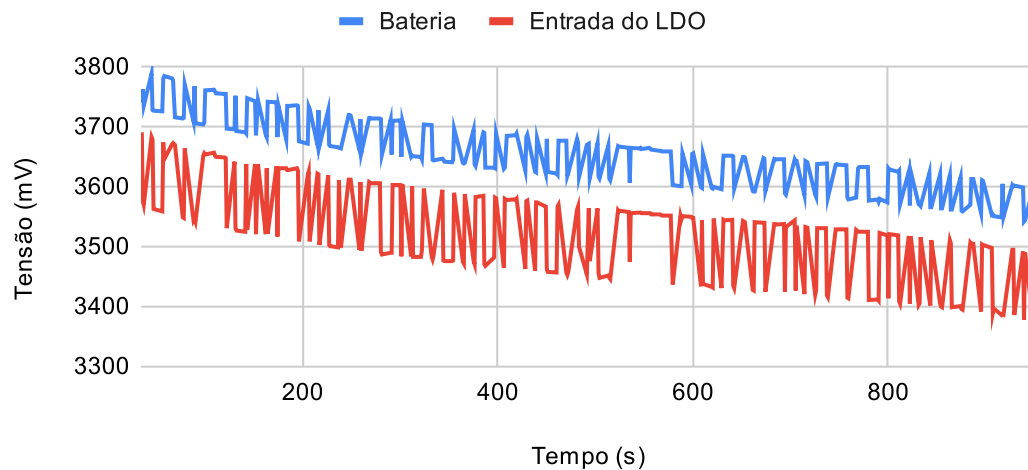
Para o treinamento da rede neural artificial foi usado o código presente no Anexo H, que lê dois arquivos diferentes, cada um com dados uma descarga completa da bateria feita em dias distintos, e usa um deles para treinamento enquanto usa o outro para validação. Ou seja, o objetivo foi descobrir se seria viável usar uma IA para estimar o estado da carga da bateria para que tal informação pudesse ser obtida de forma fácil e sem usar qualquer equação complexa. Para tornar o teste ainda mais difícil, a carga inicial de ambos os testes foi diferente para simular uma situação onde a bateria já estivesse um pouco desgastada e não tivesse o mesmo nível de carga que tinha quando nova.

4 Resultados

4.1 Monitoramento contínuo de vazão e pressão

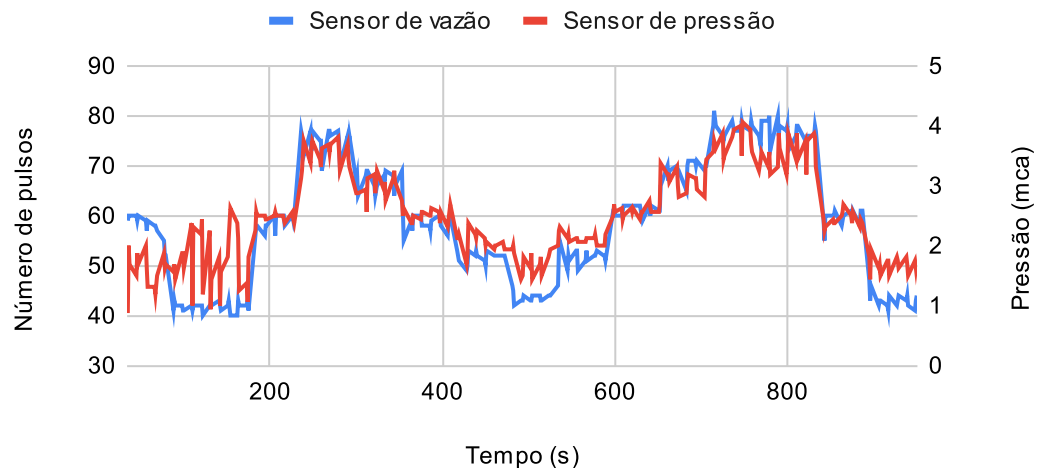
A Figura 18 apresenta o correto monitoramento de pressão e vazão com uma taxa de amostragem de 1 Hz e variações abruptas nessas grandezas por causa das variações de 5 Hz e 10 Hz na frequência da bomba feitas durante o teste, enquanto a Figura 17 apresenta uma autonomia de 15 minutos durante o teste.

Figura 17 – Tensão na bateria (mV)



Fonte: o autor

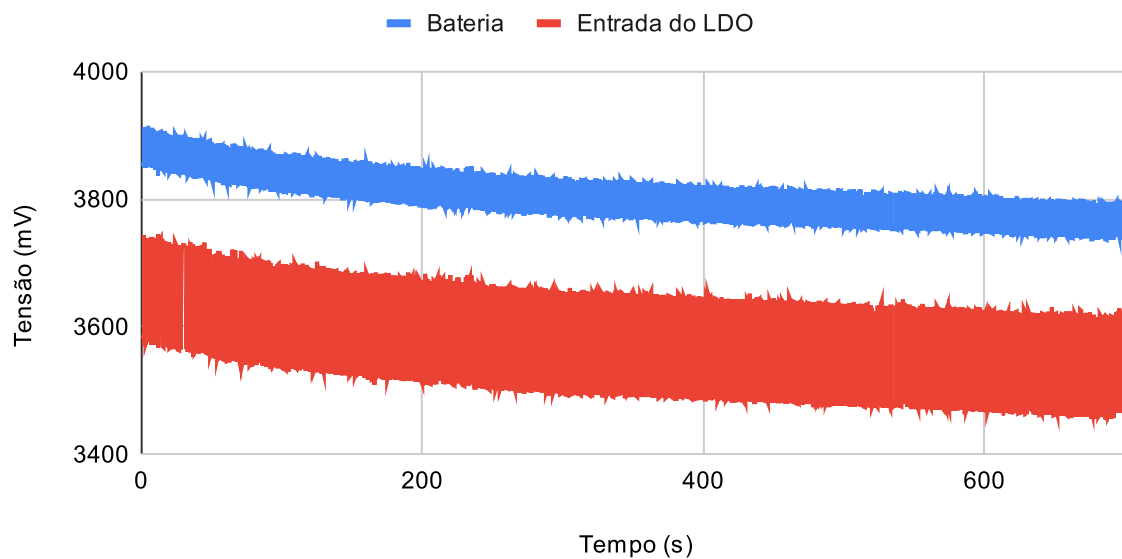
Figura 18 – Frequência de pulsos do sensor de vazão (Hz)



Fonte: o autor

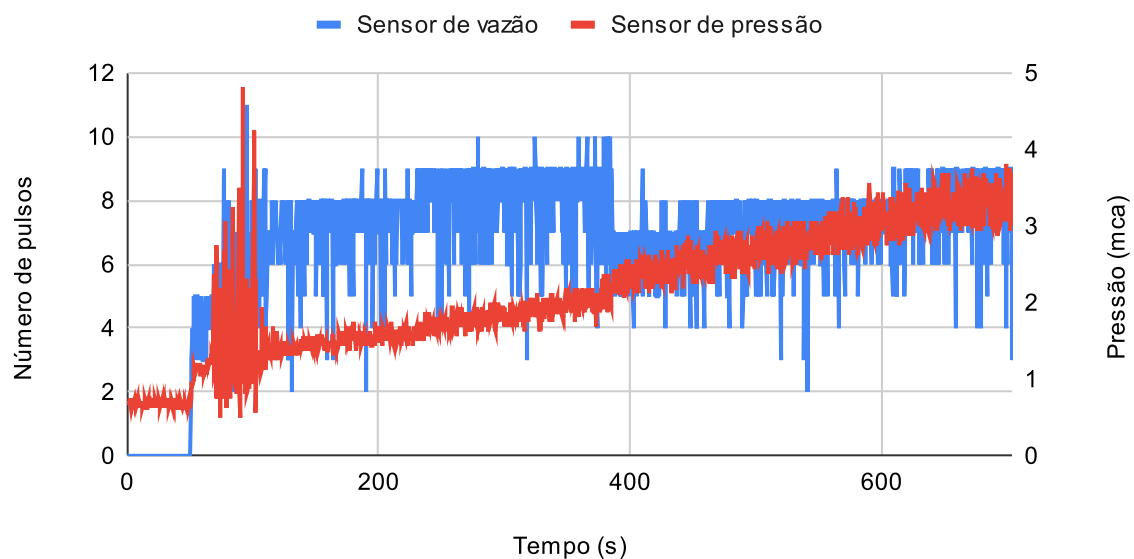
A elevação da frequência de amostragem para 10 Hz foi responsável pelas leituras do sensor de vazão pararem de ser lidas corretamente, como pode ser visto na Figura 20, enquanto o sensor de pressão funcionou corretamente e assim mostrou o efeito na pressão causado por um aumento gradual de 1 Hz da frequência da bomba a cada 30 segundos. Em relação à autonomia da bateria, ela foi um pouco menor neste teste, sendo de aproximadamente 13 minutos, como apresentado na Figura 19.

Figura 19 – Tensão na bateria (mV)



Fonte: o autor

Figura 20 – Frequência de pulsos do sensor de vazão (Hz)

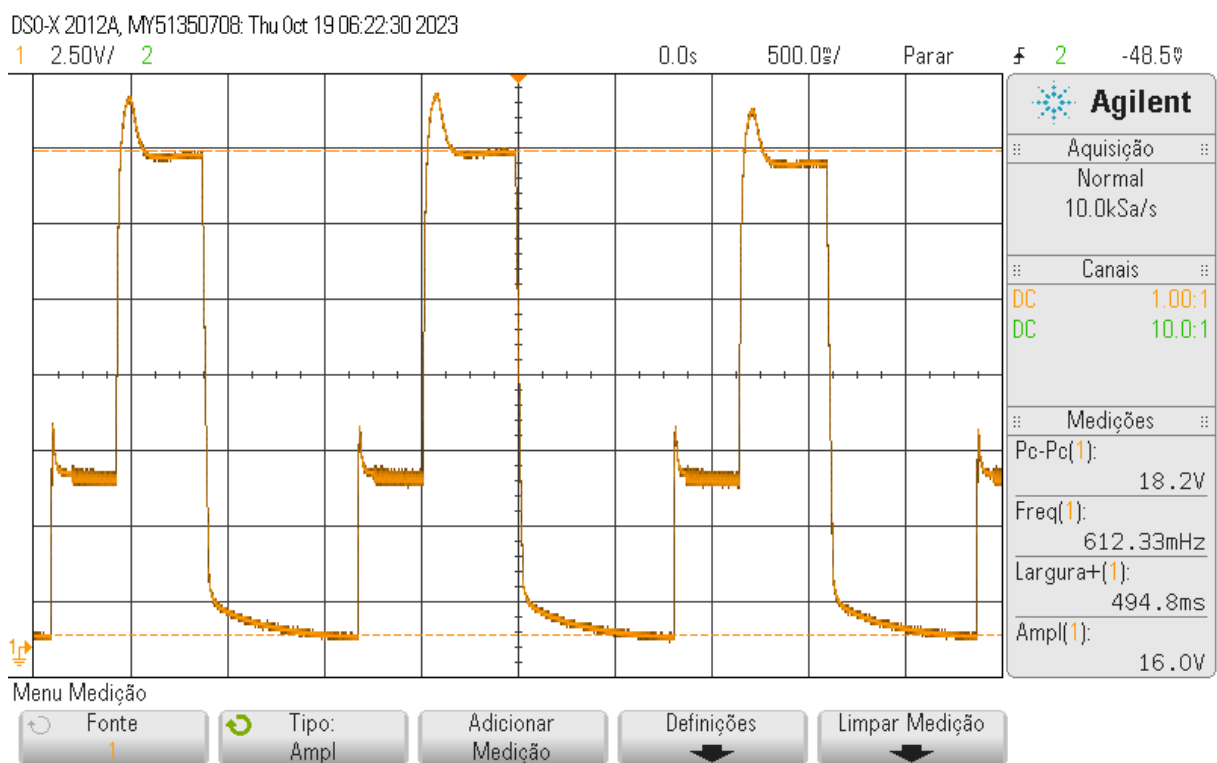


Fonte: o autor

4.2 Análise do sensor de pressão com conversor *boost*

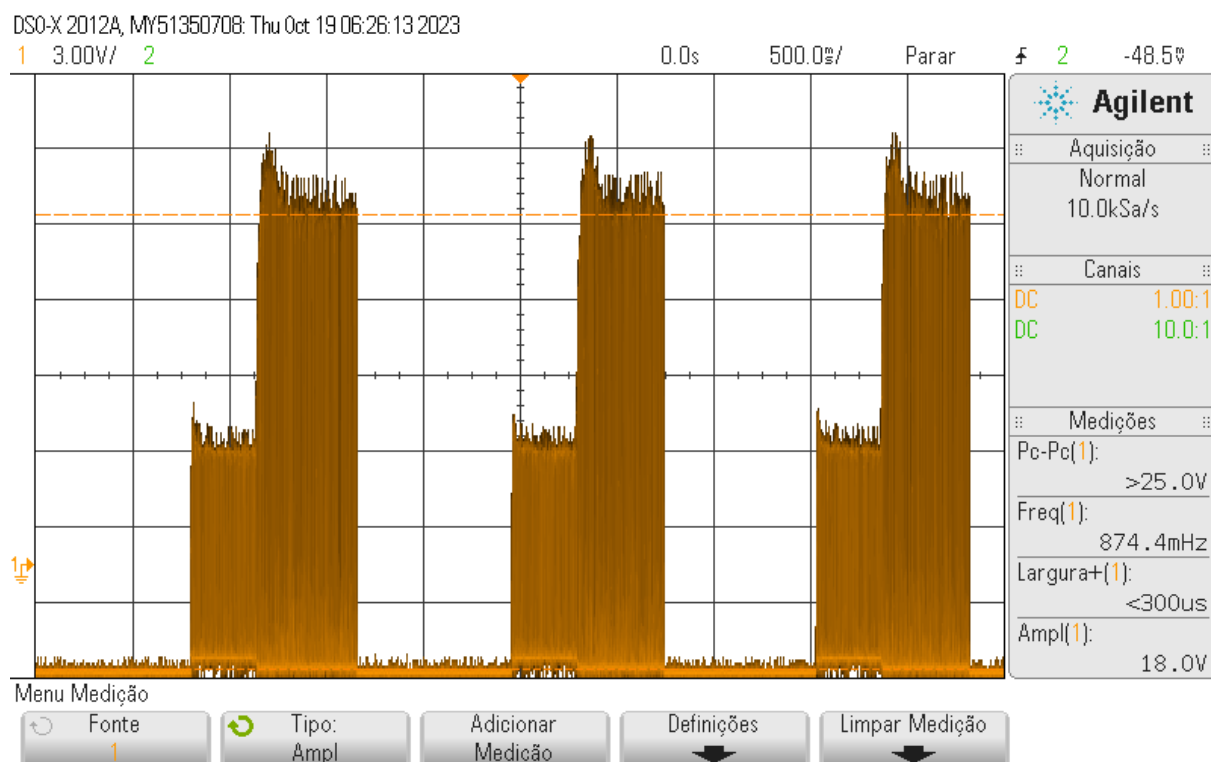
Em relação à alimentação do sensor de pressão por meio do conversor *boost* alimentado pela bateria, foi possível descobrir que a tensão no sensor se estabiliza após cerca de 500 ms, como mostrado na Figura 21, intervalo aproximadamente igual ao que pode ser mensurado na Figura 23, que apresenta uma "estabilização" na tensão do resistor após aproximadamente o mesmo intervalo tempo, porém nessa imagem é perceptível o ruído do sinal, que talvez possa ser melhorado por meio da adição de um capacitor em paralelo com o resistor, e possivelmente também tenha alguma relação com o ruído registrado na análise da tensão sobre o transistor NPN do conversor, apresentada na Figura 22, melhor detalhado nas Figuras 24 e 25, que apresentam respectivamente a tensão sobre o transistor durante os instantes iniciais de ativação do sensor de pressão e a tensão sobre o mesmo ponto após a estabilização, onde é perceptível a grande diferença na amplitude do sinal como indicado pela quarta medição, no canto inferior direito.

Figura 21 – Tensão sobre o sensor de pressão



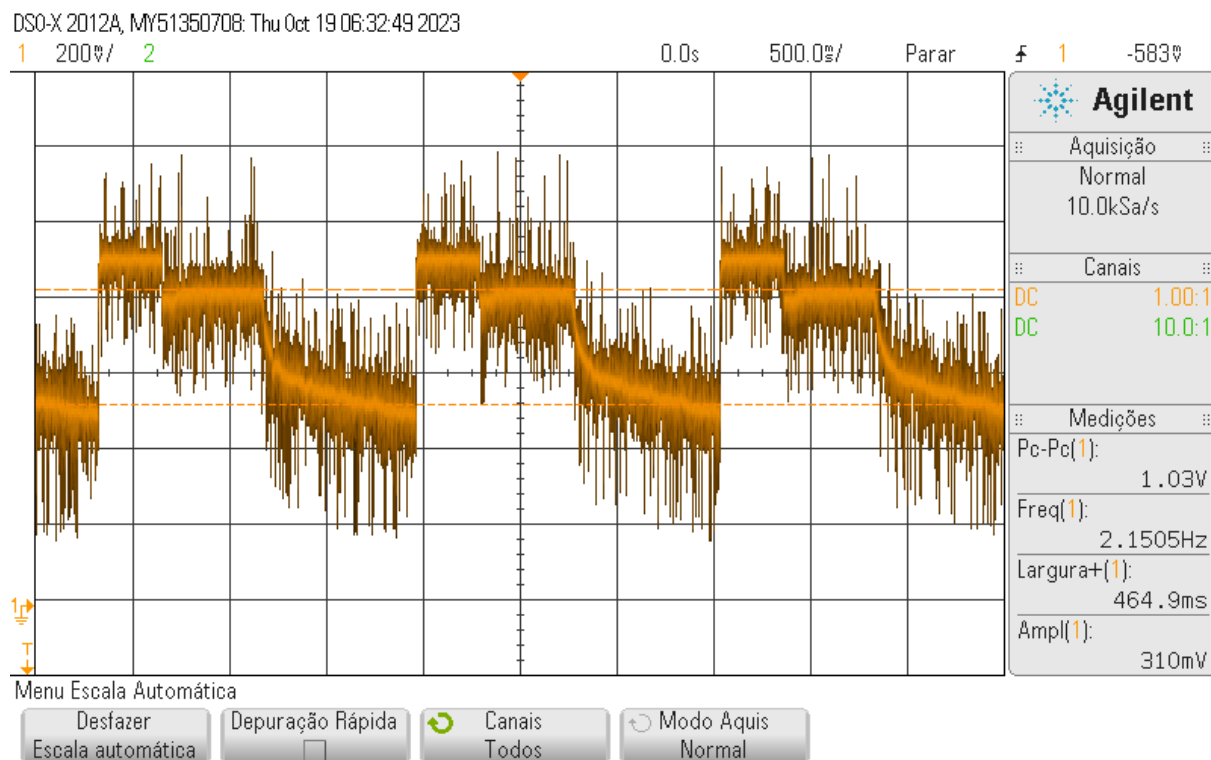
Fonte: o autor

Figura 22 – Tensão sobre o transistor NPN



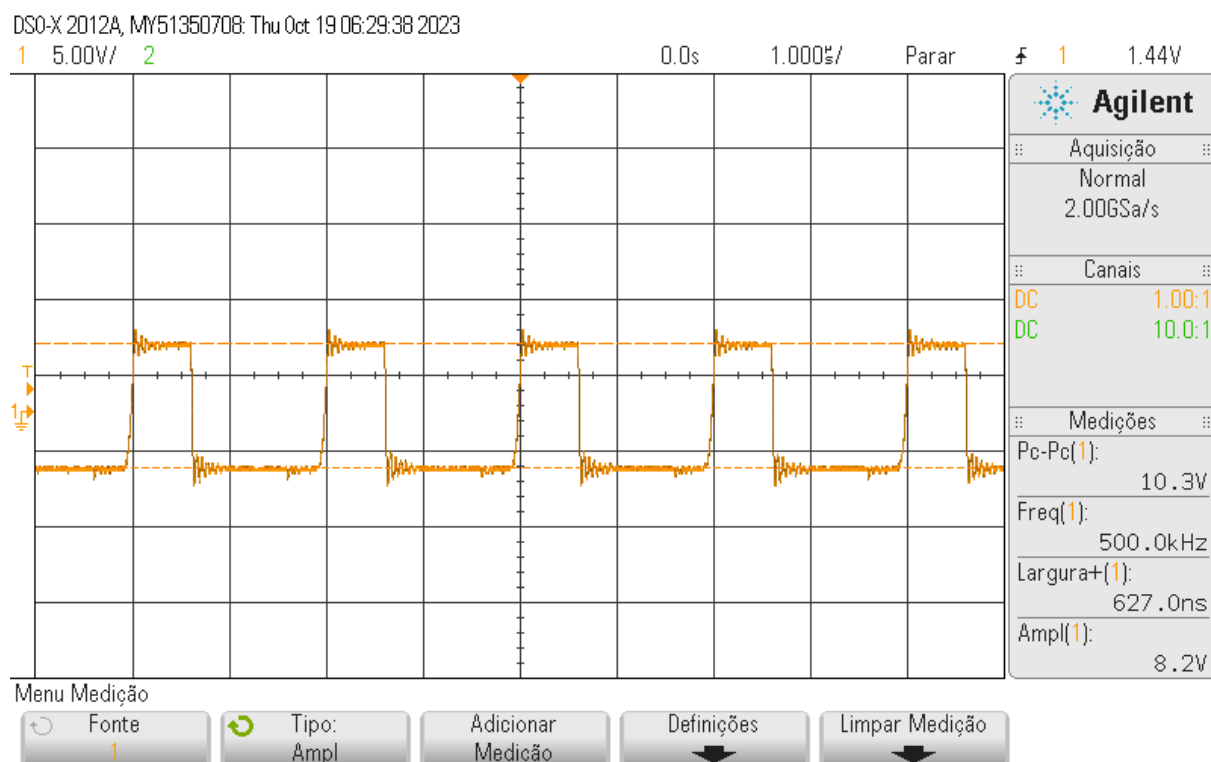
Fonte: o autor

Figura 23 – Tensão sobre o resistor do sensor de pressão



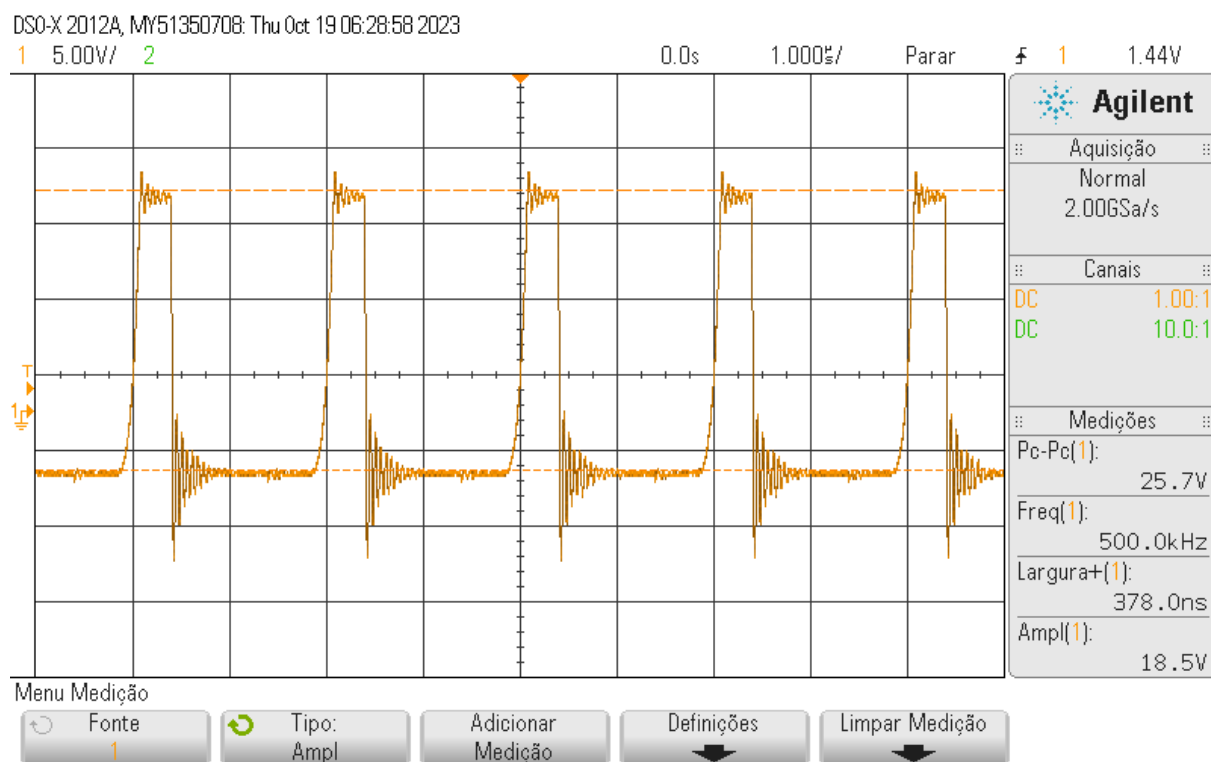
Fonte: o autor

Figura 24 – Tensão sobre o transistor durante ligamento do sensor de pressão



Fonte: o autor

Figura 25 – Tensão sobre o transistor após estabilização do sensor de pressão

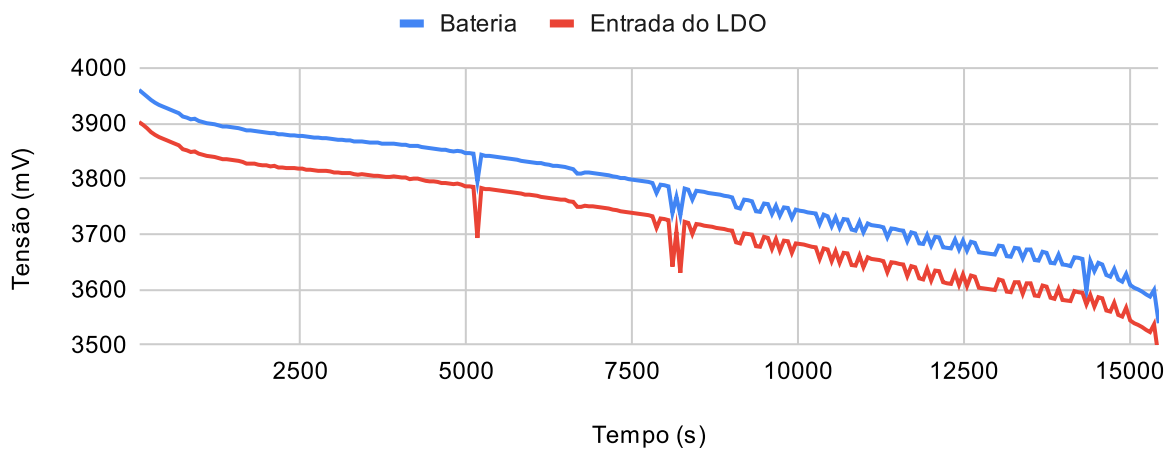


Fonte: o autor

4.3 Análise de autonomia de bateria

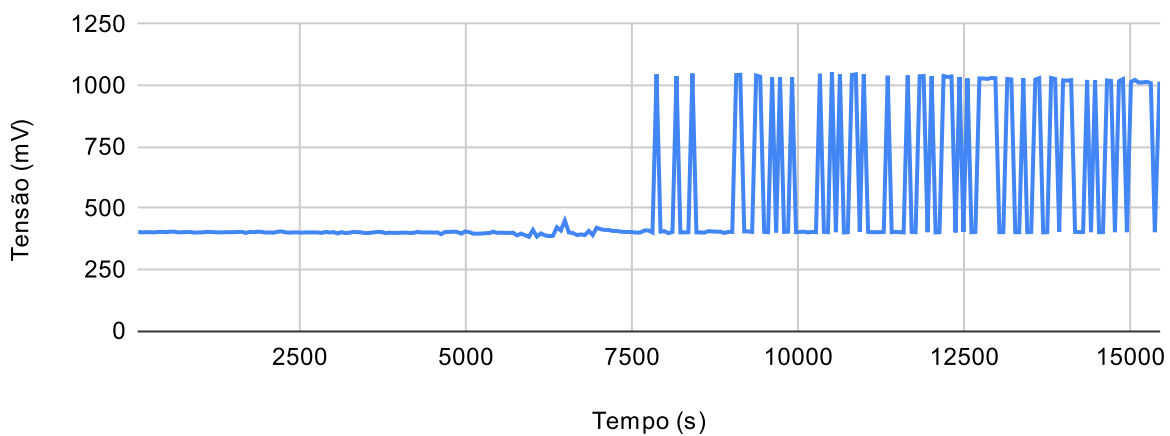
Foram feitas duas rodadas de testes, na primeira o sensor de pressão apresentou leituras incorretas abaixo de 50% de bateria, como ilustrado na Figura 27, porque ele era ativado ao mesmo tempo que o WiFi estava tentando se conectar à rede WiFi, o que acabava causando oscilações na tensão, como mostrado na Figura 26. Na segunda rodada de testes, como o sensor de pressão permanecia ativo apenas antes do WiFi ser ativado, a corrente máxima do sistema diminuiu, o que estabilizou as leituras do sensor de pressão durante longo do teste, como ilustra a Figura 29. Quanto à duração máxima da bateria, foi possível atingir cerca de 6 horas, como registrado no gráfico da Figura 28.

Figura 26 – Tensões da bateria e na entrada do LDO durante 1º teste



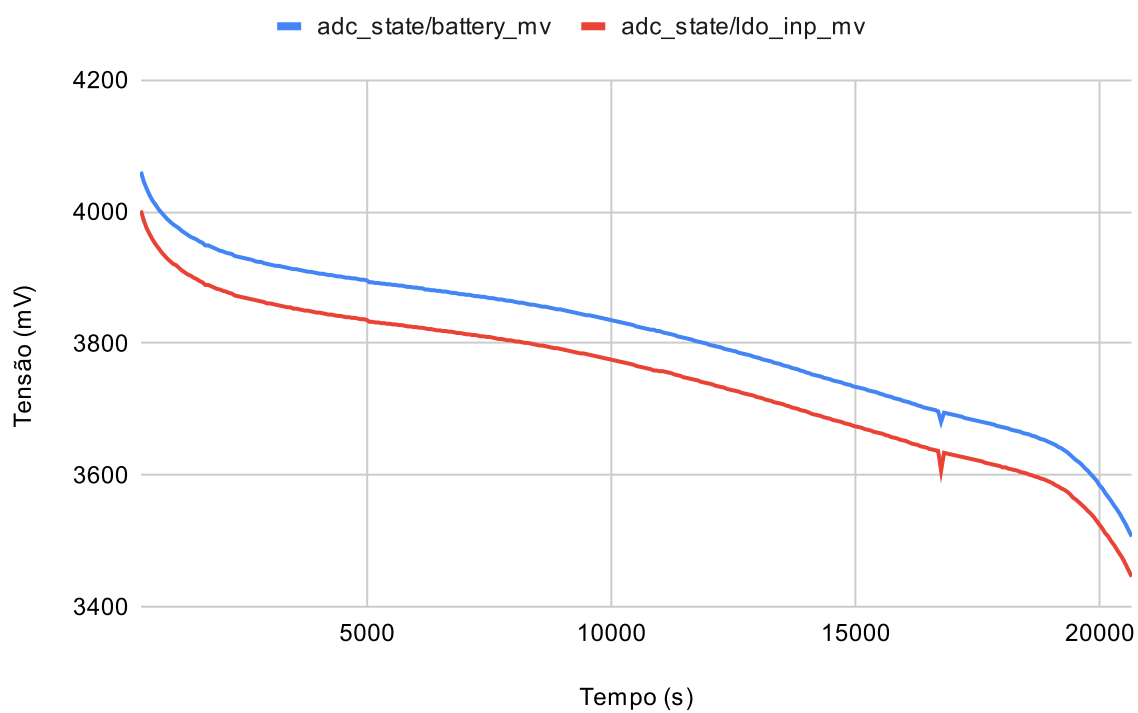
Fonte: o autor

Figura 27 – Tensão lida sobre resistor do sensor de pressão durante 1º teste



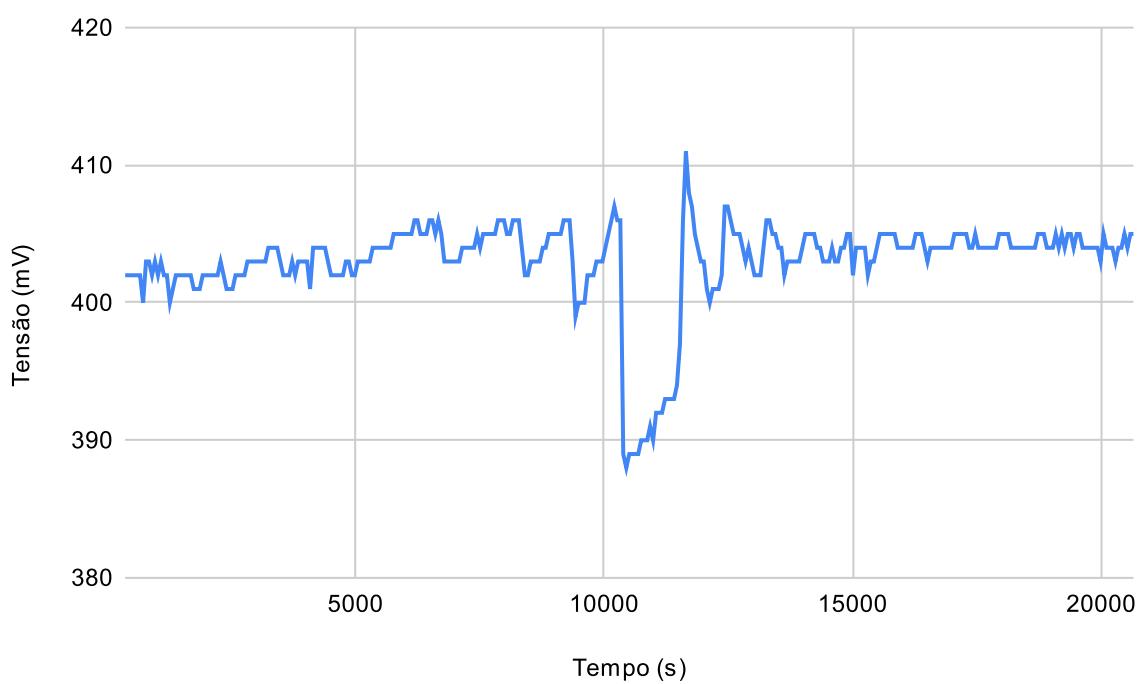
Fonte: o autor

Figura 28 – Tensão da bateria e na entrada do LDO durante 2º teste



Fonte: o autor

Figura 29 – Tensão lida sobre resistor do sensor de pressão durante 2º teste

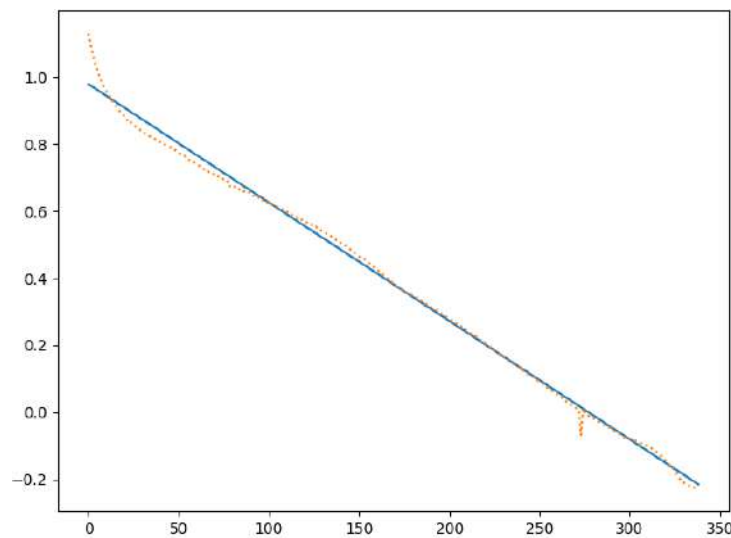


Fonte: o autor

4.4 Estimativa de porcentagem com IA

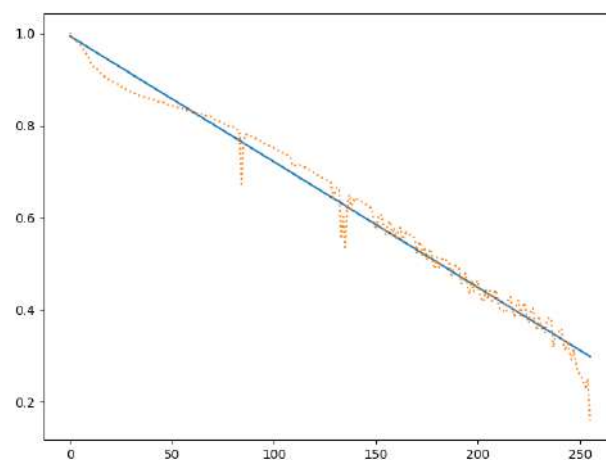
Usando os dados das duas rodadas de testes diferentes, foram registrados os resultados mostrados nas Figuras 31 e 30, cuja linha azul representa a porcentagem teórica "ideal" estimada da bateria enquanto a linha laranja tracejada representa a porcentagem estimada pela inteligência artificial apenas com os dados de tensão de bateria, tensão de entrada do LDO e última tensão de bateria lida. Ambos os testes tiveram um erro quadrático médio inferior a 1% e conseguiram fazer estimativas realistas mesmo diante de mudanças de código entre um teste e outro.

Figura 30 – Estimativa de porcentagem de bateria com validação no 1º teste



Fonte: o autor

Figura 31 – Estimativa de porcentagem de bateria com validação no 2º teste



Fonte: o autor

5 Conclusão

Neste trabalho, foram alcançados resultados notáveis na implementação de um projeto complexo, permitindo a captura de informações críticas, como pressão, vazão e grandezas elétricas, a partir do dispositivo. No entanto, vale ressaltar que este trabalho oferece uma série de vantagens distintas que merecem destaque.

Primeiramente, a abordagem de hardware adotada se destaca pela sua capacidade de ler o sensor de pressão com uma bateria simples. Apesar das limitações da bateria de baixa qualidade usada, essa abordagem permitiu a coleta de dados essenciais, evidenciando o potencial para melhorias significativas na autonomia através da adoção de fontes de energia mais eficientes.

Além disso, a escolha de desenvolver o projeto em Rust oferece diversas vantagens. A linguagem Rust é reconhecida por sua segurança em relação a erros de memória, o que é fundamental para garantir a confiabilidade do sistema em ambientes críticos. As bibliotecas disponíveis em Rust são conhecidas por sua facilidade de uso, proporcionando uma experiência de desenvolvimento mais eficaz. Adicionalmente, o desempenho otimizado do Rust contribui para a eficiência operacional do sistema, enquanto a acessibilidade a técnicas de inteligência artificial facilita a implementação de soluções avançadas.

Outro destaque importante deste projeto é a possibilidade de integração de macro e micro medições no mesmo sistema. Essa abordagem permite o cruzamento de dados entre diferentes escalas, fornecendo uma visão abrangente das condições do ambiente. Essa capacidade de combinar informações de diferentes níveis de granularidade pode ser fundamental para análises mais aprofundadas e tomada de decisões estratégicas.

Por fim, como propostas de trabalhos futuros, pode-se citar a substituição do ESP usado por outro com menor consumo, avaliar diminuir a frequência de transmissão de dados e embarcar a IA dentro do ESP para que o SOC possa ser enviado junto com as demais informações.

Referências

- ARAÚJO, J. V. S. d. *MONITORAMENTO E CONTROLE DE PRESSÃO DE REDES DE ABASTECIMENTO DE ÁGUA UTILIZANDO IOT E CONTROLE FUZZY*. Tese (Trabalho de Conclusão de Curso (Graduação)) — Universidade Federal da Paraíba, Paraíba, 2021. Disponível em: <<https://www.cear.ufpb.br/juan/wp-content/uploads/2022/07/2021-MONITORAMENTO-E-CONTROLE-DE-PRESS%C3%83O-DE-REDES-DE-ABASTECIMENTO-DE-%C3%81GUA-UTILIZANDO-IOT-E-CONTROLE-FUZZY.pdf>>. Citado na página 18.
- Brasil. SNIS. *Diagnóstico Temático - Serviços de Água e Esgoto - Visão Geral*. [S.l.], 2021. Disponível em: <http://antigo.snis.gov.br/downloads/diagnosticos/ae/2020/DIAGNOSTICO_TEMATICO_VISAO_GERAL_AE_SNIS_2021.pdf>. Citado na página 17.
- CSS, T. *Install Tailwind CSS with SvelteKit - Tailwind CSS*. 2023. Disponível em: <<https://tailwindcss.com/docs/guides/sveltekit>>. Citado na página 39.
- ESPRESSIF. *The Rust on ESP Book*. 2023. Disponível em: <<https://esp-rs.github.io/book/print.html>>. Citado na página 33.
- KIBWEN. Reddit Post, *Internet archaeology: the definitive, end-all source for why Rust is named "Rust"*. 2014. Disponível em: <www.reddit.com/r/rust/comments/27jvdt/internet_archaeology_the_definitive_endall_source/>. Citado na página 19.
- OLIVEIRA, A. H. D. *et al.* Desenvolvimento de um hidrômetro digital para a classe residencial com comunicação remota de longo alcance. In: REDIN, E. (Ed.). *Ciências Rurais em Foco – Volume 6*. Editora Poisson, 2022. ISBN 9786558661580. Disponível em: <https://www.poisson.com.br/livros/Ciencias_Rurais/volume6/Ciencias_Rurais_vol6.pdf>. Citado na página 17.
- OVERFLOW, S. *Stack Overflow Developer Survey 2023*. 2023. Disponível em: <https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023>. Citado na página 19.
- SAGA. *Manual de operações - Medidor de água ultrassônico residencial Saga*. 2022. Disponível em: <<https://www.sagamedicao.com.br/site/wp-content/uploads/2022/08/MANUAL-DE-OPERACOES-ULTRASSONICO-RESIDENCIAL-SAGA-19-08-2022.pdf>>. Citado na página 29.
- SOBRINHO, P. A. *Abastecimento De Agua*. [S.l.]: Não definido, 2003. ISBN 978-85-900823-6-1. Citado na página 24.
- VENDEMIATTI, C. *Sistema remoto para monitoramento do consumo de água em tempo real*. Tese (Dissertação (mestrado)) — Universidade Estadual Paulista, Sorocaba, 2020. Disponível em: <<https://repositorio.unesp.br/bitstreams/524d170d-bffd-4cd1-a51f-9ad12f106736/download>>. Citado na página 18.

VITE. *Vite - Getting Started*. 2023. Disponível em: <<https://vitejs.dev>>. Citado na página 39.

Anexos

ANEXO A – Manual do transdutor de pressão

PRESSGAGE - PRESSÃO

TRANSMISSOR DE PRESSÃO INDUSTRIAL TPI-PRESS



APLICAÇÕES

O transmissor de Pressão Industrial **TPI-PRESS** tem como características principais a qualidade, precisão e durabilidade. Devido sua construção em invólucro de material inoxidável, pode ser utilizado em processos e nas mais variadas aplicações, como na área de instrumentação em geral, equipamentos industriais, sistemas hidráulicos, pneumáticos, refrigeração, compressores, medição de nível, controle de vazão, indústrias de diversos segmentos (alimentos, químico, petroquímico, farmacêutico, automobilístico, saneamento dentre outros).

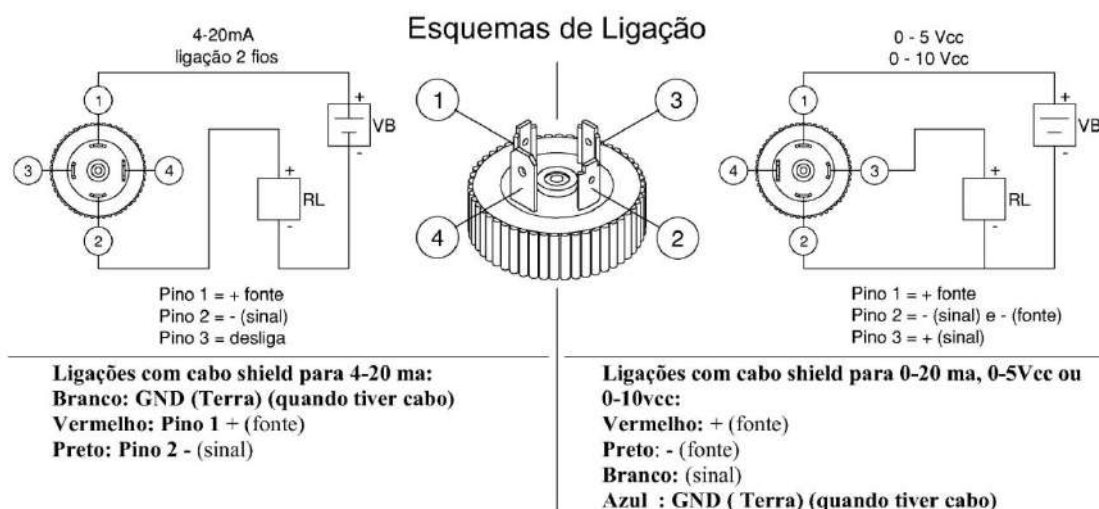
CARACTERÍSTICAS TÉCNICAS

- Material do invólucro: aço inoxidável AISI 316L
- Sensor piezorresistivo: aço inoxidável AISI 316L
- Comprimento do corpo: 35, 65 ou 85mm (a definir)
- Material do corpo: aço inoxidável 304L (316L opcional)
- Vedação anel o'ring borracha nitrílica
- Grau de proteção: IP-65 / IP-67
- Roscas: NPT ou BSP ($\frac{1}{2}$, $\frac{1}{4}$, dentre outras, a definir)
- Faixa de pressão de -1 a 1000 bar, baixa-pressão até 100 mbar (faixa de pressão e unidade de pressão, definir)
- Unidades de engenharia: Bar, mBar, MPa, psi, Kg/cm² / MCA, mmCA, mH₂O, mmHg, polH₂O, (dentre outras, a definir)
- Conector: DIN 43650
- Alimentação: 12...30 Vcc / 0...5 Vcc (diferencial opcional)
- Sinal de saída 4 a 20 mA (opcionais: 0 a 5 VCC, 0 a 10 VCC e 0 a 20 VCC ou mV/V)
- Precisão: 0,25% ou 0,5% F.E (a definir, opcional 0,1%F.E)
- Tempo de resposta = ou < 1 milissegundo
- Resolução de saída analógica infinita

PRESSGAGE - PRESSÃO

INSTALAÇÃO

O conector paralelo (BSP) e a base devem ser corretamente selados. Durante a instalação, deve ter cuidado com o sextavado, que é apertado com uma chave especial (chave de boca de 1.1/8") com um torque máximo de 30 N/m.



PRECAUÇÕES:

- Assegure-se de que, quando instalado, o transmissor de pressão não esteja sujeitado à linha excessiva de alta temperatura e pressão superior a sua faixa indicada no mesmo.

⚠ ATENÇÃO! Este transdutor de pressão é projetado para trabalhar com conexão a terra através da conexão da linha de processo e com cabo especial (shielded), para garantir maior proteção e segurança do processo.

⚠ ATENÇÃO! CUIDADO COM A MEMBRANA EXPOSTA, POIS NÃO PODE SER PRESSIONADA!

ANEXO B – Exemplo de página Web

```
1 <html>
2   <head>
3     <title>Minha Página Web</title>
4     <style>
5       h1 {
6         color: blue;
7         font-size: 24px;
8       }
9       p {
10        color: gray;
11        margin-bottom: 20px;
12      }
13    </style>
14    <script defer>
15      document.querySelector("h1")
16        .addEventListener(
17        "click",
18        () => alert("O título foi clicado!"));
19    </script>
20  </head>
21  <body>
22    <h1>Bem-vindo ao meu site!</h1>
23    <p>Esta é uma página web simples.</p>
24  </body>
25 </html>
```


ANEXO C – Exemplo de código Svelte

```
1 <script lang="ts">
2   let name: string = 'world';
3 </script>
4
5 <h1 class="text-3xl font-bold underline">
6   Hello {name}!
7 </h1>
8
9 <style lang="postcss">
10   :global(html) {
11     background-color: theme(colors.gray.100);
12   }
13 </style>
```


ANEXO D – config.toml do projeto embarcado

```
1 [target.xtensa-esp32-none-elf]
2 runner = "espflash flash --monitor"
3
4 [build]
5 rustflags = [
6     "-C", "link-arg=-Tlinkall.x",
7     "-C", "link-arg=-Trom_functions.x",
8     "-C", "link-arg=-nostartfiles",
9 ]
10 target = "xtensa-esp32-none-elf"
11
12 [unstable]
13 build-std = ["core"]
```


ANEXO E – Código embarcado no ESP

```

1  #![no_std]
2  #![no_main]
3  #![feature(type_alias_impl_trait)]
4  #![feature(generic_arg_infer)]
5  #![feature(error_in_core)]
6
7  // read about async Rust in https://rust-lang.github.io/async-book/01
   _getting_started/01_chapter.html
8  // hal code examples in https://github.com/esp-rs/esp-hal/tree/main/
   esp32c3-hal
9  // wifi code examples in https://github.com/esp-rs/esp-wifi/blob/main/
   /examples-esp32c3
10 // board repo in https://github.com/Xinyuan-LilyGO/LilyGo-T-OI-PLUS
11
12 use core::sync::atomic::{AtomicU16, Ordering};
13 use desse::{Desse, DesseSized};
14 use dotenvy_macro::dotenv;
15 use embedded_storage::{ReadStorage, Storage};
16 use embedded_svc::{
17     io::{Read, Write},
18     ipv4::Interface,
19     wifi::{ClientConfiguration, Configuration, Wifi},
20 };
21 use esp_backtrace as _;
22 use esp_println::println;
23 use esp_storage::FlashStorage;
24 use esp_wifi::{
25     current_millis, initialize,
26     wifi::{utils::create_network_interface, WifiMode},
27     wifi_interface::{Socket, WifiStack},
28     EspWifiInitFor,
29 };
30 use hal::{
31     clock::{ClockControl, Clocks},
32     gpio::{GpioPin, Input, Output, PullUp, PushPull, RTCPin, Unknown,
33            IO},
33     interrupt,
34     ledc::{LSGlobalClkSource, LowSpeed, LEDC},

```

```

35     macros::ram,
36     peripherals::{self, Peripherals},
37     prelude::*,
38     rtc_cntl::{
39         get_wakeup_cause,
40         sleep::{Ext0WakeupSource, TimerWakeupSource, WakeupLevel},
41     },
42     timer::TimerGroup,
43     Delay, Rng, Rtc,
44 };
45 use mqtttrs::{decode_slice, encode_slice, Pid};
46 use smoltcp::wire::Ipv4Address;
47
48 const FLASH_ADDR: u32 = 0x9000;
49 const INTERVAL_SEC: u64 = 60;
50
51 const SSID: &str = dotenv!("SSID");
52 const PASSWORD: &str = dotenv!("PASSWORD");
53
54 static ADITIONAL_PULSES: AtomicU16 = AtomicU16::new(0);
55
56 #[derive(serde::Serialize, Clone, Copy, Debug, Default)]
57 pub struct I2CADCRead {
58     pub battery_mv: u16,
59     pub ldo_inp_mv: u16,
60     pub pressure_mv: u16,
61 }
62
63 impl I2CADCRead {
64     pub fn read(i2c: hal::i2c::I2C<'static, hal::peripherals::I2C0>)
65         -> Self {
66         use ads1x1x::{Ads1x1x, FullScaleRange, SlaveAddr};
67         let address = SlaveAddr::default();
68         let mut adc = Ads1x1x::new_ads1115(i2c, address);
69         adc.set_full_scale_range(FullScaleRange::Within6_144V)
70             .unwrap();
71         let a0 = nb::block!(adc.read(&mut ads1x1x::channel::SingleA0)
72             ).unwrap();
73         let a1 = nb::block!(adc.read(&mut ads1x1x::channel::SingleA1)
74             ).unwrap();
75         let a3 = nb::block!(adc.read(&mut ads1x1x::channel::SingleA3)
76             ).unwrap();

```

```

73         Self {
74             battery_mv: Self::get_mv(a3),
75             ldo_inp_mv: Self::get_mv(a1),
76             pressure_mv: Self::get_mv(a0),
77         }
78     }
79
80     fn get_mv(val: i16) -> u16 {
81         if val < 0 {
82             0
83         } else {
84             (val as u32 * 3 / 16) as u16
85         }
86     }
87 }
88
89 #[derive(serde::Serialize, Clone, Copy, Debug, Default)]
90 pub struct DeviceState {
91     pub adc_state: I2CADCRead,
92     pub n_pulses: u16,
93     pub time_sec: u64,
94 }
95
96 impl DeviceState {
97     pub fn publish<'a, 'b>(
98         &self,
99         socket: &mut Socket<'a, 'b>,
100     ) -> Result<(), esp_wifi::wifi_interface::IoError> {
101         let mut buffer = [0u8; 10000];
102         let json = serde_json_core::to_string::<Self, 10000>(&self).
103         unwrap();
104         let publish = mqtttrs::Publish {
105             dup: false,
106             payload: json.as_bytes(),
107             qospid: mqtttrs::QosPid::AtLeastOnce(Pid::new()),
108             retain: false,
109             topic_name: "jaedson/tocloud",
110         };
111         let size = encode_slice(&publish.into(), &mut buffer).unwrap
112         ();
113         socket.write_all(&buffer[..size])?;
114         socket.flush()?;

```

```

113     esp_println::println!("awaiting...");
114     let len = socket.read(&mut buffer).unwrap();
115     let decoded = decode_slice(&buffer[..len]).unwrap().unwrap();
116     esp_println::println!("{decoded:?}");
117     Ok(())
118 }
119 }
120
121 #[derive(Desse, DesseSized, Default)]
122 pub struct PersistentState {
123     pub n_pulses: u16,
124     pub next_transmission: u64,
125     pub charger_state: bool,
126 }
127
128 impl PersistentState {
129     pub fn reset() -> Self {
130         let counter = PersistentState::default();
131         let bytes = counter.serialize();
132         let mut flash = FlashStorage::new();
133         flash.write(FLASH_ADDR, &bytes).unwrap();
134         counter
135     }
136
137     pub fn get(add_pulse: bool) -> Self {
138         let mut flash = FlashStorage::new();
139         let mut bytes = [0u8; _];
140         flash.read(FLASH_ADDR, &mut bytes).unwrap();
141         let mut counter: PersistentState = PersistentState::
deserialize_from(&bytes);
142         if add_pulse {
143             counter.n_pulses += 1;
144             let bytes = counter.serialize();
145             flash.write(FLASH_ADDR, &bytes).unwrap();
146         }
147         counter
148     }
149
150     pub fn next(
151         &self,
152         new_pulses: Option<u16>,
153         new_charger_state: Option<bool>,

```

```

154         rtc: &Rtc<'static>,
155     ) -> Self {
156         let mut flash = FlashStorage::new();
157         let time = rtc.get_time_ms() / 1000 + INTERVAL_SEC / 2; //
min interval of at least 50%
158         let next_transmission = time + INTERVAL_SEC - time %
INTERVAL_SEC;
159         let state = PersistentState {
160             n_pulses: match new_pulses {
161                 Some(v) => v,
162                 None => self.n_pulses,
163             },
164             next_transmission,
165             charger_state: new_charger_state.unwrap_or(self.
charger_state),
166         };
167         let bytes = state.serialize();
168         flash.write(FLASH_ADDR, &bytes).unwrap();
169         state
170     }
171 }
172
173 #[entry]
174 fn entry() -> ! {
175     let peripherals = Peripherals::take();
176     let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
177
178     let (pulse_wake, timer_wake) = match get_wakeup_cause() {
179         hal::reset::SleepSource::Ext0 => (true, false),
180         hal::reset::SleepSource::Timer => (false, true),
181         _ => (false, false),
182     };
183
184     let persistent_state = if !pulse_wake && !timer_wake {
185         PersistentState::reset()
186     } else {
187         PersistentState::get(pulse_wake)
188     };
189
190     let system = peripherals.DPORT.split();
191     let clocks = ClockControl::max(system.clock_control).freeze();
192     let mut peripheral_clock_control = system.

```

```
    peripheral_clock_control;
193
194    let rtc = Rtc::new(peripherals.RTC_CNTL);
195    let mut pulse_pin = io.pins.gpio12.into_pull_up_input();
196    {
197        let time = rtc.get_time_ms() / 1000;
198        if time < persistent_state.next_transmission {
199            // reenter in sleep mode if we still need to wait
200            go_sleep(&clocks, persistent_state.next_transmission,
pulse_pin, rtc);
201        }
202    }
203
204    // time to send data
205
206    pulse_pin.listen(hal::gpio::Event::FallingEdge);
207    interrupt::enable(peripherals::Interrupt::GPIO, interrupt::
Priority::Priority1).unwrap();
208
209    let timer_group0 = TimerGroup::new(peripherals.TIMG0, &clocks, &
mut peripheral_clock_control);
210    let mut wdt = timer_group0.wdt;
211    wdt.start(30u64.secs()); // prevent unknown stop
212
213    interrupt::enable(
214        peripherals::Interrupt::I2C_EXT0,
215        interrupt::Priority::Priority1,
216    )
217    .unwrap();
218    let i2c = hal::i2c::I2C::new(
219        peripherals.I2C0,
220        io.pins.gpio4,
221        io.pins.gpio16,
222        100u32.kHz(),
223        &mut peripheral_clock_control,
224        &clocks,
225    );
226
227    let mut ledc = LEDC::new(peripherals.LEDC, &clocks, &mut
peripheral_clock_control);
228    ledc.set_global_slow_clock(LSGlobalClkSource::APBClk);
229
```

```

230     let mut active_state_led = io.pins.gpio2.into_push_pull_output();
231     active_state_led.set_high().unwrap();
232     let mut pressure_en_pin = io.pins.gpio22.into_push_pull_output();
233     pressure_en_pin.set_high().unwrap();
234     start_boost_pwm(&ledc, io.pins.gpio23);
235     let mut delay = Delay::new(&clocks);
236     delay.delay_ms(500u32);
237     let adc_state = I2CADCRead::read(i2c);
238     pressure_en_pin.set_low().unwrap();
239     active_state_led.set_low().unwrap();
240
241     let timer = TimerGroup::new(peripherals.TIMG1, &clocks, &mut
peripheral_clock_control).timer0;
242     let rng: Rng = Rng::new(peripherals.RNG);
243     let (wifi, _) = peripherals.RADIO.split();
244     let radio_control = system.radio_clock_control;
245     let mut socket_set_entries: [smoltcp::iface::SocketStorage<'_>;
3] = Default::default();
246
247     let mut rx_buffer = [0u8; 1536];
248     let mut tx_buffer = [0u8; 1536];
249     let wifi_stack = connect_wifi(
250         timer,
251         rng,
252         wifi,
253         radio_control,
254         &clocks,
255         &mut socket_set_entries,
256     );
257     let mut socket = wifi_stack.get_socket(&mut rx_buffer, &mut
tx_buffer);
258     socket.work();
259     socket.open(Ipv4Address::new(192, 168, 0, 114).into_address(),
1883).unwrap();
260
261     let new_charger_state = control_charger(
262         io.pins.gpio15.into_push_pull_output(),
263         adc_state.battery_mv,
264         persistent_state.charger_state,
265     );
266
267     let initial_pulses = persistent_state.n_pulses;

```

```

268     let sent_n_pulses =
269         connect_to_broker_and_publish_device_state(adc_state,
initial_pulses, &mut socket, &rtc);
270
271     let additional_pulses = ADDITIONAL_PULSES.load(Ordering::Acquire);
272     let remaining_n_pulses = initial_pulses + additional_pulses -
sent_n_pulses;
273     let next = persistent_state.next(Some(remaining_n_pulses), Some(
new_charger_state), &rtc);
274     go_sleep(&clocks, next.next_transmission, pulse_pin, rtc);
275 }
276
277 fn connect_wifi<'a>(
278     timer: hal::Timer<hal::timer::Timer0<hal::peripherals::TIMG1>>,
279     rng: Rng,
280     wifi: hal::radio::Wifi,
281     radio_control: hal::system::RadioClockControl,
282     clocks: &Clocks,
283     socket_set_entries: &'a mut [smoltcp::iface::SocketStorage<'a>;
3],
284 ) -> WifiStack<'a> {
285     let init = initialize(EspWifiInitFor::Wifi, timer, rng,
radio_control, &clocks).unwrap();
286     let (iface, device, mut controller, sockets) =
287         create_network_interface(&init, wifi, WifiMode::Sta,
socket_set_entries).unwrap();
288     let wifi_stack = WifiStack::new(iface, device, sockets,
current_millis);
289
290     let client_config = Configuration::Client(ClientConfiguration {
291         ssid: SSID.into(),
292         password: PASSWORD.into(),
293         ..Default::default()
294     });
295     controller.set_configuration(&client_config).unwrap();
296     controller.start().unwrap();
297     controller.connect().unwrap();
298
299     println!("Wait to get connected");
300     loop {
301         let res = controller.is_connected();
302         match res {

```

```

303         Ok(connected) => {
304             if connected {
305                 break;
306             }
307         }
308         Err(err) => {
309             println!("{:?}", err);
310             loop {}
311         }
312     }
313 }
314 println!("{:?}", controller.is_connected());
315
316 println!("Wait to get an ip address");
317 loop {
318     wifi_stack.work();
319
320     if wifi_stack.is_iface_up() {
321         println!("got ip {:?}", wifi_stack.get_ip_info());
322         break;
323     }
324 }
325
326 wifi_stack
327 }
328
329 fn connect_to_broker_and_publish_device_state<'a, 'b>(
330     adc_state: I2CADCRead,
331     initial_pulses: u16,
332     socket: &mut Socket<'a, 'b>,
333     rtc: &Rtc<'static>,
334 ) -> u16 {
335     if let Err(err) = connect_to_mqtt_broker(socket) {
336         println!("Fail to connect to broker: {err:?}");
337         return 0;
338     }
339     let additional_pulses = ADDITIONAL_PULSES.load(Ordering::Acquire);
340     let device_state = DeviceState {
341         adc_state,
342         n_pulses: initial_pulses + additional_pulses,
343         time_sec: rtc.get_time_ms() / 1000,
344     };

```

```
345     println!("{device_state:?}");
346     if let Err(err) = device_state.publish(socket) {
347         println!("error while sending state: {err:?}");
348         return 0;
349     }
350     println!("device state sent");
351     device_state.n_pulses
352 }
353
354 #[ram]
355 #[interrupt]
356 unsafe fn GPIO() {
357     ADDITIONAL_PULSES.fetch_add(1, Ordering::Release);
358 }
359
360 fn go_sleep(
361     clocks: &Clocks<'static>,
362     next_transmission: u64,
363     wake_pin: GpioPin<Input<PullUp>, 12>,
364     mut rtc: Rtc,
365 ) -> ! {
366     let time = rtc.get_time_ms() / 1000;
367     let mut delay = Delay::new(&clocks);
368     let remaining_time = next_transmission - time;
369     let timer = TimerWakeupSource::new(core::time::Duration::
from_secs(remaining_time));
370     println!("waking up in {remaining_time} seconds");
371     let mut ext0_pin = wake_pin.into_pull_up_input();
372     let ext0 = Ext0WakeupSource::new(&mut ext0_pin, WakeupLevel::Low)
;
373     rtc.sleep_deep(&[&timer, &ext0], &mut delay);
374 }
375
376 pub fn control_charger(
377     mut charger_en_pin: GpioPin<Output<PushPull>, 15>,
378     battery_mv: u16,
379     current_state: bool,
380 ) -> bool {
381     if battery_mv > 4250 {
382         charger_en_pin.rtcio_pad_hold(false);
383         charger_en_pin.set_low().unwrap();
384         charger_en_pin.rtcio_pad_hold(true);
```

```

385     println!("disabling charger");
386     false
387 } else if battery_mv < 4100 && !current_state {
388     charger_en_pin.rtcio_pad_hold(false);
389     charger_en_pin.set_high().unwrap();
390     charger_en_pin.rtcio_pad_hold(true);
391     println!("enabling charger");
392     true
393 } else {
394     current_state
395 }
396 }
397
398 pub fn start_boost_pwm<'a>(ledc: &'a LEDC<'a>, pwm_pin: GpioPin<
    Unknown, 23>) {
399     use hal::ledc::{channel, timer};
400
401     let pwm_pin = pwm_pin.into_push_pull_output();
402
403     let mut lstimer0 = ledc.get_timer::<LowSpeed>(timer::Number::
    Timer2);
404     lstimer0
405         .configure(timer::config::Config {
406             duty: timer::config::Duty::Duty5Bit,
407             clock_source: timer::LSClockSource::APBCLK,
408             frequency: 500u32.kHz(),
409         })
410         .unwrap();
411
412     let mut channel0 = ledc.get_channel(channel::Number::Channel0,
    pwm_pin);
413     channel0
414         .configure(channel::config::Config {
415             timer: &lstimer0,
416             duty_pct: 50,
417             pin_config: channel::config::PinConfig::PushPull,
418         })
419         .unwrap();
420 }
421
422 fn connect_to_mqtt_broker<'a, 'b>(
423     socket: &mut Socket<'a, 'b>,

```

```
424 ) -> Result<(), esp_wifi::wifi_interface::IoError> {
425     let mut buffer = [0u8; 1024];
426     let connect = mqtttrs::Connect {
427         clean_session: true,
428         client_id: "esp32",
429         keep_alive: 120,
430         last_will: None,
431         password: None,
432         username: None,
433         protocol: mqtttrs::Protocol::MQTT311,
434     };
435     let size = encode_slice(&connect.into(), &mut buffer).unwrap();
436     socket.write_all(&buffer[..size])?;
437     socket.flush()?;
438
439     let mut buffer = [0u8; 512];
440     socket.read(&mut buffer).unwrap();
441     let decoded = decode_slice(&buffer).unwrap().unwrap();
442     esp_println::println!("{decoded:?}");
443     Ok(())
444 }
```

ANEXO F – Código do servidor

```

1  async fn serve_dir() {
2      use std::net::SocketAddr;
3      use tower_http::services::ServeDir;
4
5      let app =
6          axum::Router::new().nest_service("/", ServeDir::new("./"));
7      let addr = SocketAddr::from([0, 0, 0, 0], 8080);
8
9      axum::Server::bind(&addr)
10         .serve(app.into_make_service())
11         .await
12         .unwrap();
13 }
14
15 fn process_msg(
16     link_rx: &mut rumqttc::local::LinkRx,
17     file: &mut std::fs::File,
18     first: &mut bool,
19 ) {
20     use std::io::{Seek, Write};
21     let notification = match link_rx.recv().unwrap() {
22         Some(v) => v,
23         None => return,
24     };
25     if let rumqttc::Notification::Forward(forward) = notification {
26         let payload = forward.publish.payload;
27         file.seek(std::io::SeekFrom::End(-1)).unwrap();
28         if *first {
29             *first = false;
30         } else {
31             file.write(",\n".as_bytes()).unwrap();
32         }
33         file.write(&payload).unwrap();
34         file.write("]".as_bytes()).unwrap();
35         file.flush().unwrap();
36     }
37 }
38

```

```
39 #[tokio::main]
40 async fn main() {
41     use std::io::{Seek, Write};
42     tokio::spawn(serve_dir());
43     let config = config::Config::builder()
44         .add_source(config::File::with_name("rumqttd.toml"))
45         .build()
46         .unwrap()
47         .try_deserialize()
48         .unwrap();
49     let mut broker = rumqttd::Broker::new(config);
50     let (mut link_tx, mut link_rx) =
51         broker.link("singlenode").unwrap();
52     std::thread::spawn(move || broker.start().unwrap());
53     link_tx.subscribe("#").unwrap();
54     let mut file = std::fs::File::create("./result.json").unwrap();
55     file.write("[]".as_bytes()).unwrap();
56     file.seek(std::io::SeekFrom::Start(0)).unwrap();
57     let mut first = true;
58     loop {
59         process_msg(&mut link_rx, &mut file, &mut first)
60     }
61 }
```

ANEXO G – Código do aplicativo Web

```

1 <script lang="ts">
2   import * as mqtt from 'mqtt/dist/mqtt.min'
3   import { LineChart } from 'chartist'
4   import { onMount } from 'svelte'
5
6   interface Message {
7     adc_state: {
8       battery_mv: number
9       ldo_inp_mv: number
10      pressure_mv: number
11    }
12    n_pulses: number
13    time_sec: number
14  }
15
16  let last_values: Message[] = []
17  let chart_ldo_inp_mv: LineChart
18  let chart_battery_mv: LineChart
19  let chart_n_pulses: LineChart
20  let chart_pressure_mv: LineChart
21
22  function getLabel(msg: Message, index: number) {
23    if (index % 10) return ''
24    return new Date(msg.time_sec)
25      .toLocaleTimeString()
26      .split(':')
27      .filter((_, i) => i > 0)
28      .join(':')
29  }
30
31  onMount(() => {
32    const default_data = { labels: [], series: [[]] }
33    chart_ldo_inp_mv = new LineChart('#ldo_inp_mv', default_data)
34    chart_battery_mv = new LineChart('#battery_mv', default_data)
35    chart_n_pulses = new LineChart('#n_pulses', default_data)
36    chart_pressure_mv = new LineChart('#pressure_mv', default_data)
37  })
38

```

```

39  const client = mqtt.connect('ws://54.80.140.156:8083')
40  client.on('connect', () => client.subscribe('#', console.error))
41  client.on('message', (_topic: string, message: string) => {
42    const msg = JSON.parse(message) as Message
43    if (last_values.length > 100) last_values.shift()
44    msg.time_sec = new Date().valueOf()
45    last_values.push(msg)
46    const labels = last_values.map(getLabel)
47    chart_ldo_inp_mv.update({
48      labels,
49      series: [last_values.map((v) => v.adc_state.ldo_inp_mv)],
50    })
51    chart_battery_mv.update({
52      labels,
53      series: [last_values.map((v) => v.adc_state.battery_mv)],
54    })
55    chart_n_pulses.update({
56      labels,
57      series: [last_values.map((v) => v.n_pulses)],
58    })
59    chart_pressure_mv.update({
60      labels,
61      series: [last_values.map((v) => v.adc_state.pressure_mv / 100)
62    ],
63    })
64  })
65  </script>
66  <main class="p-8 flex gap-2 flex-col">
67    <h1 class="text-3xl font-bold text-center mb-2">
68      MONITORAMENTO DE PRESSÃO E VAZÃO UTILIZANDO IOT E CLOUD
69    </h1>
70    <section class="flex gap-2">
71      <div class="flex flex-col flex-1">
72        <h3 class="text-center">Tensão de entrada do LD0 (mV)</h3>
73        <div class="ct-chart w-full h-56" id="ldo_inp_mv" />
74      </div>
75      <div class="flex flex-col flex-1">
76        <h3 class="text-center">Tensão da bateria (mV)</h3>
77        <div class="ct-chart w-full h-56" id="battery_mv" />
78      </div>
79    </section>

```



```
80 <section class="flex gap-2">
81   <div class="flex flex-col flex-1">
82     <h3 class="text-center">Pulsos do sensor de vazão</h3>
83     <div class="ct-chart w-full h-56" id="n_pulses" />
84   </div>
85   <div class="flex flex-col flex-1">
86     <h3 class="text-center">Corrente no sensor de pressão (mA)</h3>
87     <div class="ct-chart w-full h-56" id="pressure_mv" />
88   </div>
89 </section>
90 <section class="flex justify-center">
91   <a
92     class="bg-red-500 font-bold text-xl uppercase py-4 px-8 rounded
93     -lg text-white"
94     href="http://54.80.140.156/result.json"
95     download="dados.json"
96   >
97     Baixar resultados
98   </a>
99 </section>
100 </main>
```


ANEXO H – Código de treinamento da IA

```

1 import json
2 from sklearn.model_selection import train_test_split
3 import tensorflow as tf
4 from tensorflow import keras
5
6 with open('data1.json', 'r') as file:
7     data0 = json.load(file)
8
9 X_train = []
10 y_train = []
11
12 last_value = None
13 for item in data0:
14     print(item)
15     adc_state = item.get('adc_state')
16     if adc_state:
17         battery_mv = (adc_state.get('battery_mv') - 3300) / 800
18         if last_value == None:
19             last_value = battery_mv
20             continue
21         ldo_in_mv = (adc_state.get('ldo_in_mv') - 3300) / 800
22         time_sec = item.get('time_sec')
23         X_train.append([battery_mv, ldo_in_mv, last_value])
24         last_value = battery_mv
25         percentage = (17000 - time_sec) / 17000
26         y_train.append(percentage)
27
28 X_test = []
29 y_test = []
30
31 with open('data0.json', 'r') as file:
32     data1 = json.load(file)
33
34 last_value = None
35 for item in data0:
36     print(item)
37     adc_state = item.get('adc_state')
38     if adc_state:

```

```
39     battery_mv = (adc_state.get('battery_mv') - 3300) / 800
40     if last_value == None:
41         last_value = battery_mv
42         continue
43     ldo_in_mv = (adc_state.get('ldo_in_mv') - 3300) / 800
44     time_sec = item.get('time_sec')
45     X_test.append([battery_mv, ldo_in_mv, last_value])
46     last_value = battery_mv
47     percentage = (17000 - time_sec) / 17000
48     y_test.append(percentage)
49
50 # Construir o modelo
51 model = keras.Sequential([
52     keras.layers.Dense(3, activation='relu', input_shape=(3,)),
53     keras.layers.Dense(8, activation='relu'),
54     keras.layers.Dense(3, activation='relu'),
55     keras.layers.Dense(1, activation='linear')
56 ])
57
58 model.compile(optimizer='adam', loss='mean_squared_error')
59
60 # Treinar o modelo
61 model.fit(X_train, y_train, epochs=256, batch_size=32,
62         validation_data=(X_test, y_test))
63
64 # Avaliar o modelo
65 loss = model.evaluate(X_test, y_test)
66 print(f'Loss: {loss}')
67
68 import matplotlib.pyplot as plt
69
70 y_pred = model.predict(X_test)
71
72 # Crie um gráfico de dispersão (scatter plot) para comparar os
73     valores reais (y_test) com as previsões (y_pred)
74 plt.figure(figsize=(8, 6))
75 plt.plot(y_test)
76 plt.plot(y_pred, linestyle = 'dotted')
77 plt.title('Comparação entre Valores Reais e Previsões do Modelo')
78 plt.show()
```